

有限元方法及其程序设计

Terence Yu

MATLAB version: \geq R2015a

目 录

1 网格生成与数据结构	1
1 网格的图示与标记	2
1.1 网格与解的图示	2
1.2 网格的标记	9
2 辅助数据结构与几何量	14
2.1 辅助数据结构	14
2.2 网格相关的几何量	19
2.3 auxstructure 与 auxgeometry 函数	20
2.4 边界设置	22
2.4.1 边界边的定向	22
2.4.2 边界的设置	23
3 三角网格的生成	26
3.1 Delaunay 三角剖分	26
3.1.1 Delaunay 三角剖分的定义	26
3.1.2 Bowyer 逐点插入法	27
3.1.3 程序整理	32
3.2 DistMesh	33
3.2.1 DistMesh 的网格优化思想	33
3.2.2 符号距离函数	34
3.2.3 程序说明	35
3.2.4 简化版程序	38
3.2.5 添加 mesh size function	39
3.3 DistMesh3D	41

II 经典问题的有限元方法	42
4 一维问题的有限元方法	43
4.1 单元与整体的关系	43
4.1.1 整体节点基与局部节点基	43
4.1.2 整体刚度矩阵与单元刚度矩阵的关系	44
4.2 刚度矩阵与载荷向量的装配	46
4.2.1 载荷向量的装配	46
4.2.2 刚度矩阵的装配	47
4.2.3 指标装配法	48
4.2.4 sparse 装配法	50
刚度矩阵的装配	50
载荷向量的装配	53
4.3 程序设计	56
4.3.1 问题说明	56
4.3.2 刚度矩阵和载荷向量的装配	57
4.3.3 边界项的处理	58
4.3.4 程序整理	61
函数文件	61
5 Poisson 方程的边值问题	64
5.1 一些说明	64
5.1.1 问题描述与网格数据	64
5.1.2 二维问题的装配	66
指标装配法	66
sparse 装配法	70
5.2 Poisson 方程的一阶有限元方法	70
5.2.1 刚度矩阵的计算	70
5.2.2 载荷向量的计算	72
5.2.3 边界条件的处理	74
等效拉平法	74
5.2.4 边界积分的装配	76
边界单元的积分计算	77
边界条件的程序实现	78

5.2.5 程序整理	80
5.2.6 误差分析	81
5.3 Poisson 方程的二阶有限元方法	84
5.4 Poisson 方程的三阶有限元方法	86
5.5 Poisson 方程的 Crouzeix-Raviart 非协调元方法	87
6 线弹性边值问题	91
6.1 线弹性边值问题简介	91
6.1.1 问题说明	91
6.1.2 连续变分问题	92
6.1.3 有限元方法	94
6.2 刚度矩阵与载荷向量的装配	94
6.2.1 单元的向量法分析	94
6.2.2 sparse 装配指标	96
6.2.3 双线性分量的配对	97
6.3 Traction 形式的变分问题	99
6.3.1 刚度矩阵的计算	99
6.3.2 载荷向量的计算	101
6.3.3 边界条件的处理	102
6.3.4 程序整理	104
6.4 Navier 形式的变分问题	106
6.5 线弹性问题的无闭锁有限元方法	110
6.5.1 Navier 形式的变分问题	110
6.5.2 Traction 形式的变分问题	112
7 Kirchhoff 板弯问题	114
7.1 变分问题	114
7.1.1 平衡方程与边界条件	114
7.1.2 变分问题	115
7.1.3 有限元方法	115
7.2 非协调 Morley 元	116
7.2.1 Morley 元的构造	116
局部节点基	117
整体节点基	119

7.2.2	自由度的方向处理	119
	方法一: 边的符号化	119
	方法二: 符号刚度矩阵和符号载荷向量	120
7.2.3	刚度矩阵与载荷向量的计算	121
	双线性形式第一项的计算	121
	双线性形式第二项的计算	123
	载荷向量的计算	124
7.2.4	边界条件的处理	124
7.2.5	数值结果	125
7.3	非协调 Zienkiewicz 元	126
7.3.1	Zienkiewicz 元的构造	127
7.3.2	sparse 装配指标	128
7.3.3	双线性形式第一项的计算	128
7.3.4	双线性形式第二项的计算	132
7.3.5	边界条件的处理	133
7.4	非协调 Adini 元	133
7.4.1	Adini 元的构造	134
7.4.2	刚度矩阵和载荷向量的计算	135
7.5	双调和方程的混合元方法	137
7.5.1	混合元的变分问题	137
7.5.2	装配指标	138
7.5.3	刚度矩阵和载荷向量的计算	138
7.5.4	边界条件的处理	139
7.5.5	程序整理	140
III	基于变分形式的有限元程序设计	142
8	基于变分形式的有限元程序设计	143
8.1	程序的设计思路	143
8.1.1	一些重要观察	143
	观察一: 检验函数的前后问题	143
	观察二: 双线性形式计算的归结	143
	观察三: 双线性形式的装配	144

8.1.2 变分形式与程序的对应	145
8.1.3 网格信息 Th	146
8.2 assem2d 函数	147
8.2.1 变分形式的计算	148
8.2.2 变分形式的程序设计	153
8.3 int2d 函数	156
8.3.1 组合型配对的处理	157
8.3.2 双线性形式的计算	159
8.3.3 线性形式的计算	161
8.4 assem1d 和 int1d 函数	162
8.4.1 变分形式的计算	162
8.4.2 assem1d 函数	165
8.4.3 int1d 函数	169
8.5 Dirichlet 边界条件的处理	171
9 基于变分形式编程的有限元程序示例	174
9.1 一维问题 Lagrange 有限元	174
9.1.1 一维问题的一阶 Lagrange 有限元	174
9.1.2 一维问题的高阶 Lagrange 元	177
9.2 二维问题 Lagrange 有限元	179
9.2.1 二维问题的一阶 Lagrange 有限元	179
程序编写说明	179
程序整理	183
主程序	184
9.2.2 二维问题的高阶 Lagrange 元	187
9.3 线弹性问题的分块编程	190
9.3.1 第三种形式的变分问题	190
程序编写说明	190
程序整理	191
主程序	193
9.3.2 第二种形式的变分问题	194
程序编写说明	194
程序整理	196

主程序	197
9.4 双调和方程混合元方法的分块编程	198
9.5 线弹性边值问题的向量编程	200
9.5.1 函数文件	200
9.5.2 主程序	201
9.6 双调和方程混合元方法的向量编程	202
9.7 Stokes 方程	204
9.7.1 变分问题	204
9.7.2 分块表示	206
9.7.3 分块编程	207
9.7.4 向量编程	210
10 基于变分形式的三维有限元的程序设计	213
10.1 assem3d	213
10.1.1 节点基与 Gauss 求积公式	213
11 基于变分形式的有限元迭代格式的程序设计	215
11.1 发展方程	215
11.1.1 抛物型方程	215
11.1.2 双曲型方程	215
11.2 非线性问题	215
IV 有限元求解器	216
12 自适应有限元方法	217
12.1 跳量积分计算的程序设计	217
12.1.1 积分节点的连通性	217
12.1.2 插值的计算	219
12.1.3 跳量与跳量积分的计算	220
12.2 Poisson 方程的自适应有限元方法	221
12.2.1 后验误差估计	221
12.2.2 加密或标记准则	223
12.2.3 Poisson 方程的自适应有限元程序	223

12.2.4 误差指示子的计算	226
残差的计算	226
边界跳量的计算方法一	227
边界跳量的计算方法二	230
indicator 函数	231
12.3 标记算法的实现	233
12.4 最新点二分	234
12.4.1 局部加密方式	235
12.4.2 最新点二分的简单说明	237
12.4.3 标记二分的程序实现	239
12.4.4 协调二分的程序实现	242
12.4.5 Newest-node bisection 程序整理	242
12.5 板弯问题 Morley 元的自适应有限元方法	244
12.5.1 后验误差估计	244
12.5.2 边上函数跳量的计算	244
12.5.3 边上法向导数跳量的计算	246
13 均匀加密网格上的多重网格法	248
13.1 嵌套有限元	248
13.1.1 嵌套有限元空间与子空间方程	248
13.1.2 延长、限制算子与延长、限制矩阵	249
13.1.3 Galerkin 条件	251
13.2 MG 的基本思想	251
13.2.1 残差校正与频率分量	251
13.2.2 经典迭代法对频率分量的影响	253
13.2.3 两网格方法	255
13.3 MG 的算法描述	257
13.3.1 MG 的两网格添加	257
13.3.2 V-循环的伪代码	259
13.4 多重网格法的转移矩阵	260
13.4.1 延长矩阵	260
13.4.2 转移矩阵的加密获取	263
13.4.3 转移矩阵的粗化获取	264

13.5 一维问题的 MG 方法	265
13.5.1 刚度矩阵和载荷向量	265
13.5.2 多重网格函数 mgVcycle.m	267
mgVcycle 函数	267
Vcycle 函数	268
smoother 函数	268
13.5.3 主程序	269
13.6 二维问题的 MG 方法	270
13.6.1 归结为一维问题	270
13.6.2 转移矩阵的加密获取	271
13.6.3 转移矩阵的粗化获取	272
13.6.4 主程序	273
13.7 向量有限元与高阶元的 MG 方法	274
13.7.1 向量有限元的 MG 方法	274
13.7.2 二次和三次 Lagrange 元的 MG 方法	274
13.7.3 高次元 MG 使用线性元传递的原因	277
14 自适应网格上的多重网格法	279
14.1 最新点二分加密的网格粗化算法	279

Part I

网格生成与数据结构

第一章 网格的图示与标记

待整理 (改为仅适用三角形和矩形)

1.1 网格与解的图示

基本数据结构

采用陈龙编写的 iFEM 工具箱中的数据结构, 用 `node` 表示节点坐标, `elem` 表示单元的连通性, 即单元顶点编号. 考虑下图中 L 形区域的一个简单剖分, iFEM 的网页说明链接如下:

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/meshbasicdoc.html>

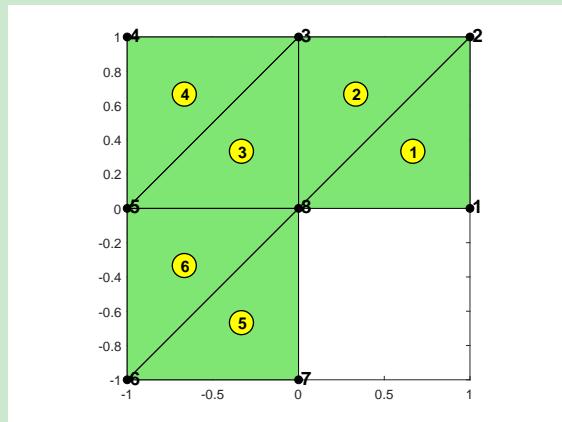


图 1.1. L 形区域的剖分

1. 矩阵 `node`: 节点坐标

编程中需要每个节点的坐标, 用 `node` 记录, 它是两列的矩阵, 第一列表示各节点的横坐标, 第二列表示各节点的纵坐标, 行的索引对应节点编号. 图中给出的顶点坐标信息如下

	1	2
1	1	0
2	1	1
3	0	1
4	-1	1
5	-1	0
6	-1	-1
7	0	-1
8	0	0

图 1.2. L-型区域的剖分

这里左侧的序号对应节点的整体编号.

2. 矩阵 `elem`: 连通性

`elem` 给出每个三角形的顶点编号, 它给出的是单元的连通性信息, 每行对应一个单元.

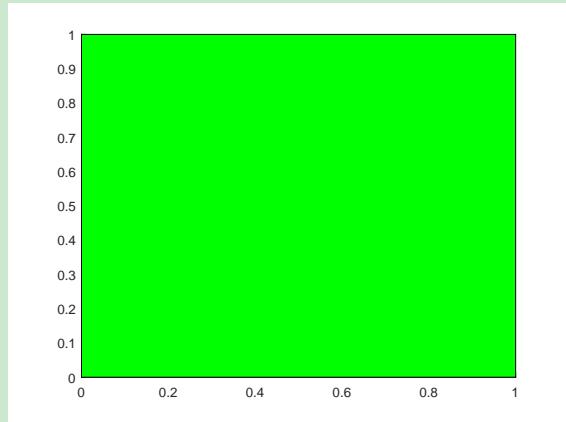
	1	2	3
1	1	2	8
2	3	8	2
3	8	3	5
4	4	5	3
5	7	8	6
6	5	6	8

图中第 i 列表示所有三角形的第 i 个点的编号, 其中 $i = 1, 2, 3$. 注意三角形顶点的顺序符合逆时针定向. `elem` 是有限元编程装配过程中 P1-元的局部整体对应.

矩形剖分的 `elem` 类似获得.

补片函数 `patch`

MATLAB 中采用补片函数 `patch` 绘制多边形, 其内置的三角剖分画图函数 `trisurf` 也是如此. 以下考虑二维多角形剖分的图示, 命名为 `showmesh.m`. 一个简单的例子如下图



该矩形如下绘制

```

1 node = [0 0; 1 0; 1 1; 0 1];
2 elem = [1 2 3 4];
3 patch('Faces',elem,'Vertices',node,'FaceColor','g')

```

对多个相同类型的单元, showmesh.m 如下编写:

```

1 function showmesh(node,elem)
2 h = patch('Faces',elem, 'Vertices', node);
3 set(h, 'facecolor',[0.5 0.9 0.45], 'edgecolor','k');
4 axis equal; axis tight;

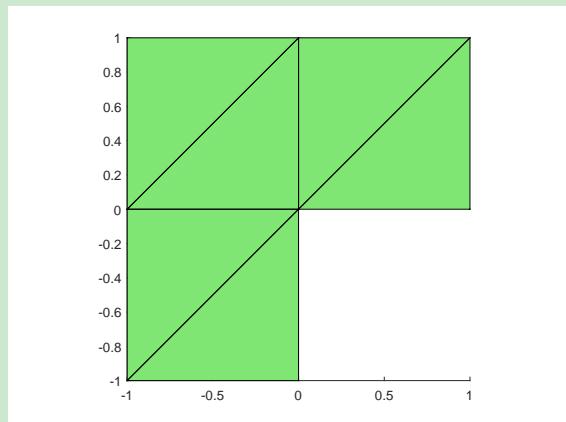
```

例 1.1 (三角剖分) 对前面的梯形区域, 如下调用 showmesh 函数

```

1 node = [1,0; 1,1; 0,1; -1,1; -1,0; -1,-1; 0,-1; 0,0];
2 elem = [1,2,8; 3,8,2; 8,3,5; 4,5,3; 7,8,6; 5,6,8];
3 showmesh(node,elem);

```

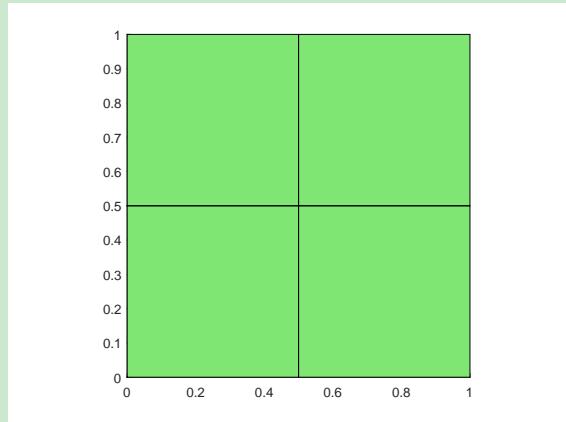


例 1.2 (四边形剖分) 对矩形区域的四边形剖分, 如下调用 showmesh 函数

```

1 [X,Y] = ndgrid(0:0.5:1,0:0.5:1);
2 node = [X(:), Y(:)];
3 elem = [1 2 5 4; 2 3 6 5; 4 5 8 7; 5 6 9 8];
4 showmesh(node,elem)

```



操作 cell 数组的 cellfun 函数

对含有不同多角形单元的区域, 因每个单元顶点数不同, elem一般以 cell 数组存储. 为了使用 `patch` 画图(避免循环语句), 需要将 elem 的每个 cell 用 `NaN` 填充成相同维数的向量, 该填充不会影响画图. 先介绍 MATLAB 中操作 cell 数组的函数 `cellfun.m`. 例如, 考虑下面的例子.

例 1.3 计算 cell 数组中元素的平均值和维数

```

1 C = {1:10, [2; 4; 6], []};
2 averages = cellfun(@mean, C)
3 [nrows, ncols] = cellfun(@size, C)
4
5 % 结果为 averages = 5.5000    4.0000      NaN
6 %      nrows = 1 3 0,      ncols = 10 1 0

```

`cellfun` 的直接输出规定为数值数组, 如果希望输出的是多种类型的元素, 那么需要指定 `UniformOutput` 为 `false`, 例如

例 1.4 对字符进行缩写

```

1 days = {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'};
2 abbrev = cellfun(@(x) x(1:3), days, 'UniformOutput', false)

```

这里的 `UniformOutput` 也可简写为 `un`, 当然 `false` 也可写为 `0`. 正因为此时输出类型任意, MATLAB 默认仍保存为 `cell` 类型. 上面的结果为

```

abbrev =
1×5 cell array
{'Mon'}    {'Tue'}    {'Wed'}    {'Thu'}    {'Fri'}

```

showmesh 函数的建立

现在考虑下图所示的剖分

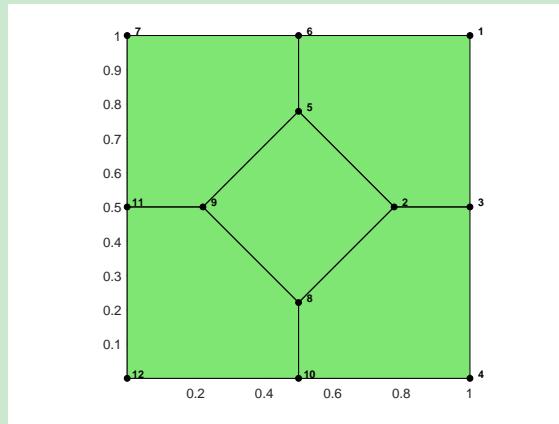


图 1.3. 多角形网格

相关的网格数据保存在 meshex1.mat 中. 程序如下

```
1 load('meshex1.mat'); % node, elem
2
3 max_n_vertices = max(cellfun(@length, elem));
4 % function to pad the vacancies (横向拼接)
5 padding_func = @(vertex_ind) [vertex_ind, ...
6     NaN(1,max_n_vertices-length(vertex_ind))];
7 tpad = cellfun(padding_func, elem, 'UniformOutput', false);
8 tpad = vertcat(tpad{:});
9 h = patch('Faces', tpad,'Vertices', node);
10 set(h, 'facecolor',[0.5 0.9 0.45], 'edgecolor', 'k');
11 axis equal; axis tight;
```

最终给出的 showmesh 函数如下

CODE 1.1. showmesh.m (绘制 2D 网格图)

```
1 function showmesh(node,elem)
2
3 dim = size(node,2);
4
5 % ----- Triangulation -----
6 % 2D
7 if ~iscell(elem) && dim==2
8     h = patch('Faces', elem, 'Vertices', node);
9 end
10
11 % ----- Polygonal mesh -----
12 if iscell(elem)
13     if iscell(elem{1}), elem = vertcat(elem{:}); end
```

```

14     max_n_vertices = max(cellfun(@length, elem));
15     padding_func = @(vertex_ind) [vertex_ind, ...
16         NaN(1,max_n_vertices-length(vertex_ind))]; % function to pad the vacancies
17     tpad = cellfun(padding_func, elem, 'UniformOutput', false);
18     face = vertcat(tpad{:}); % polygon
19     h = patch('Faces', face, 'Vertices', node);
20 end
21
22 facecolor = [0.5 0.9 0.45];
23 set(h, 'facecolor', facecolor, 'edgecolor', 'k');
24 axis equal; axis tight;

```

showsolution 函数的建立

showsolution 函数绘制解的网格图, 程序如下

CODE 1.2. showsolution.m

```

1 function showsolution(node,elem,u)
2
3 dim = size(node,2);
4
5 % ----- Triangulation -----
6 data = [node,u];
7 if ~iscell(elem) && dim==2
8     patch('Faces', elem, ...
9         'Vertices', data, ...
10        'FaceColor', 'interp', ...
11        'CData', u / max(abs(u)) );
12 end
13
14 % ----- Polygonal mesh -----
15 if iscell(elem)
16     max_n_vertices = max(cellfun(@length, elem));
17     padding_func = @(vertex_ind) [vertex_ind, ...
18         NaN(1,max_n_vertices-length(vertex_ind))]; % function to pad the vacancies
19     tpad = cellfun(padding_func, elem, 'UniformOutput', false);
20     tpad = vertcat(tpad{:});
21     patch('Faces', tpad, ...
22         'Vertices', data, ...
23         'FaceColor', 'interp', ...
24         'CData', u / max(abs(u)) );
25 end
26 axis('square');
27 sh = 0;
28 xlim([min(node(:,1))-sh, max(node(:,1))+sh])
29 ylim([min(node(:,2))-sh, max(node(:,2))+sh])
30 zlim([min(u)-sh, max(u)+sh])
31 xlabel('x'); ylabel('y'); zlabel('u');

```

```
32  
33 view(3); grid on; % view(150,30);
```

简单说明一下.

- `patch` 也可以画空间中的直面, 此时只要把 '`Vertices`' 处的数据换为三维的顶点坐标.
- 对解 u , 显然 `data = [node, u]` 就是画图的三维点坐标.
- `patch` 后的

```
'FaceColor', 'interp', 'CData', u / max(abs(u))
```

是三维图形的颜色, 它根据 '`CData`' 数据进行插值获得 (不对颜色进行设置, 默认为黑色). 也可以改为二维的

```
set(h, 'facecolor', [0.5 0.9 0.45], 'edgecolor', 'k');
```

此时显示的只是一种颜色, 但一般希望解有颜色的变化.

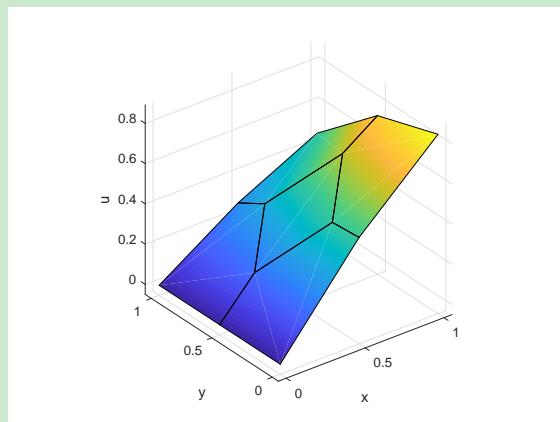
- 需要注意的是, 即便是三维数据, 若不加最后的

```
view(3); grid on; %view(150,30);
```

给出的也是二维图 (投影, 即二维剖分图).

例 1.5 如下画 $u(x, y) = \sin x \cos y$ 的图像

```
1 load('meshhex1.mat');  
2 x = node(:,1); y = node(:,2); u = sin(x).*cos(y);  
3 showsolution(node, elem, u);
```



1.2 网格的标记

节点标记

如下给出图 1.3 中的节点编号

```
1 load meshhex1.mat
2 showmesh(node,elem);
3 findnode(node);
```

函数文件如下

CODE 1.3. findnode.m

```
1 function findnode(node,range)
2 %Findnode highlights nodes in certain range.
3
4 hold on
5 dotColor = 'k.';
6 if nargin==1
7     range = (1:size(node,1))';
8 end
9 plot(node(range,1),node(range,2),dotColor, 'MarkerSize', 15);
10 shift = [0.015 0.015];
11 text(node(range,1)+shift(1),node(range,2)+shift(2),int2str(range), ...
12      'FontSize',8,'FontWeight','bold'); % show index number
13 hold off
```

当然也可简单改动以适用于三维情形.

单元标记

现在标记单元, 为此需给出单元重心, 从而标记序号. 重心使用 polycentroid.m 计算.

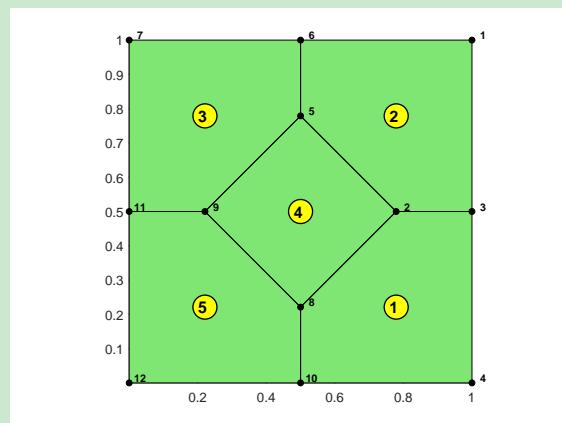


图 1.4. Polygonal mesh

主程序如下

```
1 load('meshhex1.mat');
2 showmesh(node,elem);
3 findnode(node);
4 findelem(node,elem);
```

单元标记函数如下

CODE 1.4. findelem.m

```
1 function findelem(node,elem,varargin)
2
3 hold on
4
5 NT = size(elem,1);
6 if ~iscell(elem) % transform to cell
7     elem = mat2cell(elem,ones(NT,1),length(elem(1,:)));
8 end
9
10 range = 1:NT;
11 if nargin==3, range = unique(varargin{1}); end
12 range = range(:);
13
14 center = zeros(length(range),2);
15 s = 1;
16 for iel = range(:)' % only valid for row vector
17     index = elem{iel};
18     V = node(index, :);
19     center(s,:) = polycentroid(V);
20     s = s+1;
21 end
22 plot(center(:,1),center(:,2),'o','LineWidth',1,'MarkerEdgeColor','k',...
23 'MarkerFaceColor','y','MarkerSize',18);
24 text(center(:,1)-0.01,center(:,2),int2str(range),'FontSize',12,...
25 'FontWeight','bold','Color','k');
26
27 hold off
28
29 end
```

注 1.1 这里用圆圈标记单元, 对不同的剖分, 圆圈内的数字不一定在合适的位置, 需要手动调整偏移量. 为了方便, 可直接用红色数字标记单元序号.

edge 的生成

为了标记边, 先要生成边的数据结构 edge. iFEM 对应的介绍见如下网页

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html>

该网页给出了一些辅助网格数据结构 (三角剖分), 其中的 `edge` 记录每条边的顶点编号 (去除重复边). 以下设 NT 表示三角形单元的个数, NE 表示边的个数 (不重复), 并简单说明一下那里的思路.

- 只要给出每条边两端的节点编号. 内部边在 `elem` 中会出现两次, 边界边只会出现一次, 为此可用 2 标记内部边, 1 标记边界边.
- 内部边在 `elem` 中会出现两次, 但它们是同一条边. 为了给定一致的标记, 规定每条边起点编号小于终点编号, 即 $\text{edge}(k, 1) < \text{edge}(k, 2)$.
- 对三角剖分, 通常规定三角形的第 i 条边对应第 i 个顶点. 例如, 设第 1 个三角形顶点顺序为 [1,4,5], 那么边的顺序应是 4-5, 5-1, 1-4. 在 MATLAB 中, 有如下对应

所有单元的第 1 条边: `elem(:, [2,3]); % NT * 2`

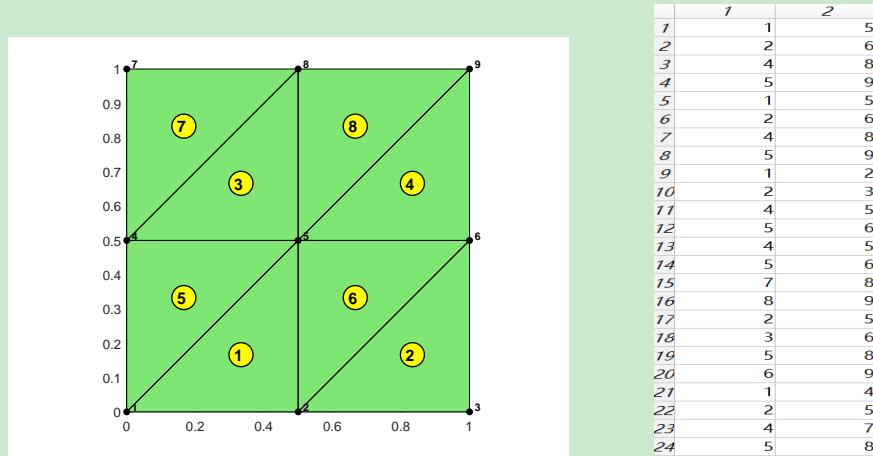
所有单元的第 2 条边: `elem(:, [3,1]); % NT * 2`

所有单元的第 3 条边: `elem(:, [1,2]); % NT * 2`

为了满足 $\text{edge}(k, 1) < \text{edge}(k, 2)$, 只需将每行的两个元素排序. 在 MATLAB 中用 `sort(A, 2)` 实现. 把这些边逐行排在一起, 则所有的边 (包含重复) 为

```
1 totalEdge = sort([elem(:, [2,3]); elem(:, [3,1]); elem(:, [1,2])], 2);
```

它是 $3NT * 2$ 的矩阵. `totalEdge` 见下面的右图.



- 设 e_{ij} ($i < j$) 表示起点 i 终点 j 的边的重数, 对应的矩阵可用 `sparse` 函数如下生成

```
1 sparse(totalEdge(:,1), totalEdge(:,2), 1)
```

注意, 在 MATLAB 中, `sparse` 有一个特殊的性质 (summation property), 即当某个位置指标出现两次, 则相应的值会相加.

- 显然, `sparse` 命令产生的矩阵, 第一行对应起点 1 的边的重数, 第二行对应起点 2 的边的重数, 等等. 希望按下述方式排列边: 先排所有起点为 1 的边, 再排所有起点为 2 的边, 等等. 由于 `find` 是按列找非零元素, 因此要把矩阵进行转置, 即

```
1 sparse(totalEdge(:,2),totalEdge(:,1),1)
```

这样, 第一列对应的是起点为 1 的边, 第二列对应的是起点为 2 的边.

- `edge` 如下给出

```
1 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
2 edge = [j,i];
```

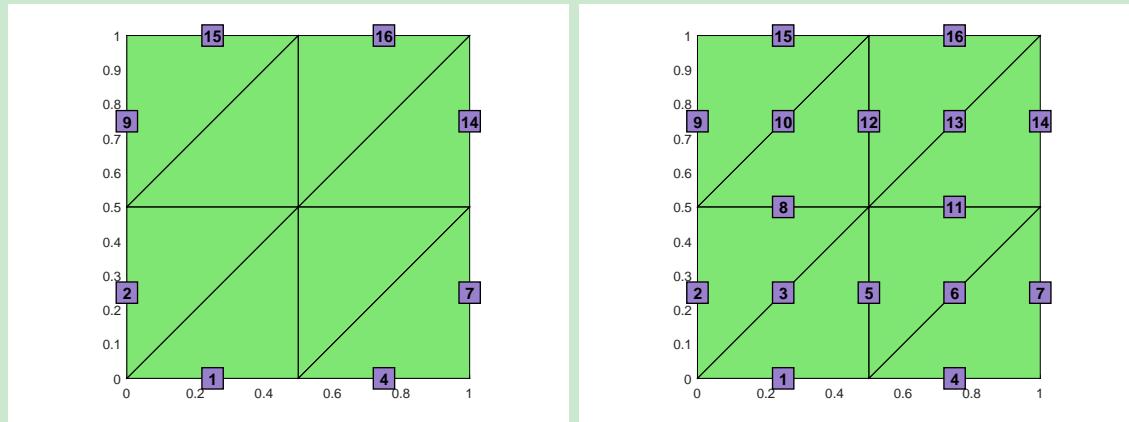
注意因为前面进行了转置, 所以 `edge = [j,i]`. 边界边为

```
1 bdEdge = edge(s==1,:);
```

- 也可直接去重复实现, 即

```
1 edge = unique(totalEdge,'rows');
```

边的标记



有了 `edge`, 边可用中点进行标记.

```
1 midEdge = (node(edge(range,1),:)+node(edge(range,2),:))/2;
2 plot(midEdge(:,1),midEdge(:,2),'s','LineWidth',1,'MarkerEdgeColor','k',...
3      'MarkerFaceColor',[0.6 0.5 0.8],'MarkerSize',20);
4 text(midEdge(:,1)-0.015,midEdge(:,2),int2str(range),...
5      'FontSize',12,'FontWeight','bold','Color','k');
```

上面的过程用函数 `findedge.m` 实现, 如

```

1 [node,elem] = squaremesh([0 1 0 1],0.5);
2 figure,
3 showmesh(node,elem);
4 bdInd = 1;
5 findedge(node,elem,bdInd);% only show boundary edges
6 figure,
7 showmesh(node,elem);
8 findedge(node,elem); % show all edges

```

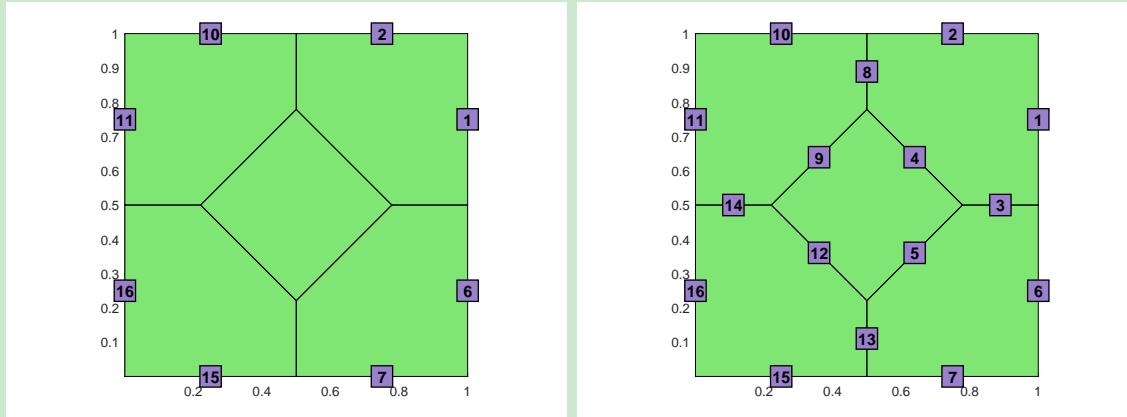
对三角形剖分, 单元上始终假设第 i 个顶点的对边为第 i 条边. 对四边形或更一般的多角形剖分, 单元上从第一个点开始逆时针对边进行局部编号. 现在将上面的思想用于多角形剖分 (四边形剖分统一为多角形剖分). 注意, 因边数不同, 要逐个单元存储每条边. 图 1.4 中第 1 个单元的顶点顺序为 [8,10,4,3,2], 我们按顺序 8-10, 10-4, 4-3, 3-2, 2-8 给出单元的边的标记. 所有边的起点就是 elem 中元素按列拉直给出的结果, 而终点就是对 [10,4,3,2,8] 这种循环的结果拉直. 可如下实现

```

1 shiftfun = @(verts) [verts(2:end),verts(1)];
2 T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
3 v0 = horzcat(elem{:})'; % the starting points of edges
4 v1 = horzcat(T1{:})'; % the ending points of edges
5 totalEdge = sort([v0,v1],2);

```

其他过程与三角剖分一致.



第二章 辅助数据结构与几何量

网格中有许多数据在计算中很有用, 例如边的标记、单元的直径、面积等. 本文参考 iFEM 给出一些辅助数据结构, 相关说明见如下网页

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html>

而且仅给出三角剖分的说明.

2.1 辅助数据结构

数据结构包括

表 2.1. 数据结构

node, elem	基本数据结构
elem2edge	边的自然序号 (单元存储)
edge	一维边的端点标记
bdEdge	边界边的端点标记
edge2elem	边的左右单元
neighbor	目标单元边的相邻单元

elem2edge 的生成

elem2edge 按单元记录每条边的自然序号, 这里同时会给出 edge, bdEdge.

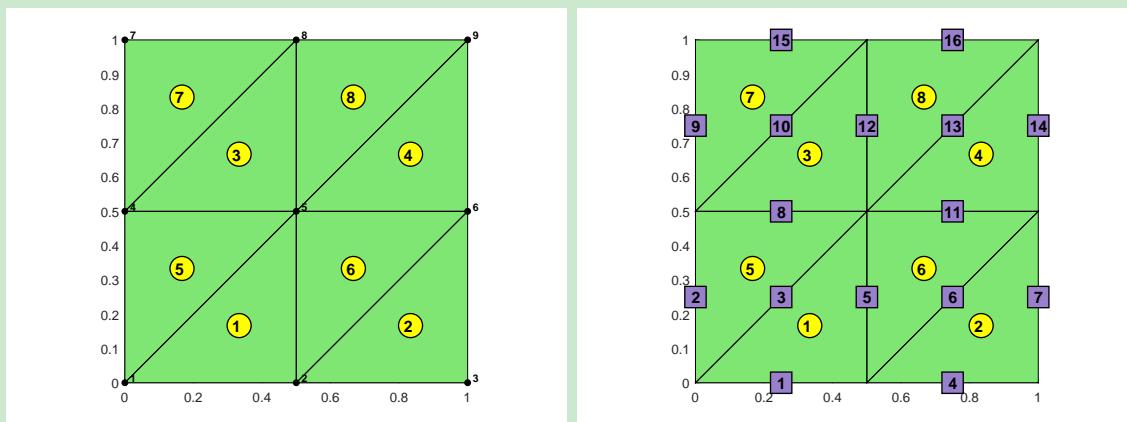


图 2.1. 三角剖分边的自然序号

用图 2.1 中给出的三角剖分说明一下 iFEM 中的思路 (有所改动).

- 根据前面的说明, 可给出含重复边的数组 `totalEdge` 见图 2.2(a).

1	2
1	5
2	6
3	8
4	9
5	1
6	2
7	4
8	5
9	1
10	2
11	4
12	5
13	4
14	5
15	7
16	8
17	2
18	3
19	5
20	6
21	1
22	2
23	4
24	5

1	2
1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	4
9	4
10	4
11	5
12	5
13	5
14	6
15	6
16	7
17	8
18	8
19	9
20	9
21	9
22	7
23	8
24	8

1
9
21
1
10
17
2
2
18
11
23
3
12
12
11
8
14
15
16
5
7
19
12
20
14
21
2
5
9
16
12

1	3
2	6
3	10
4	13
5	3
6	6
7	10
8	13
9	1
10	4
11	8
12	11
13	8
14	11
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

(a) totalEdge (b) edge (c) i1 (d) totalJ

图 2.2. elem2edge 说明

- 去除重复的行, 即得边的数据结构 `edge` (重复边一致化才能使用)

```
[edge, i1, totalJ] = unique(totalEdge, 'rows');
```

这里, `edge` 是 $NE \times 2$ 的矩阵, 对应边的集合, 注意 `unique` 会按第一列从小到大给出边 (相应地第二列也进行了排序), 见图 2.2 (b).

`i1` 是 $NE \times 1$ 的数组, 它记录 `edge` 中的每条边在原来的 `totalEdge` 的位置 (重复的按第一次出现记录). 比如, 上面的 1-5 边, 第一次出现的序号是 1, 则 `i1` 第一个元素就是 1.

`totalJ` 记录的是 `totalEdge` 的每条边在 `edge` 中的自然序号. 比如, 1-5 在 `edge` 中是第 3 个, 则 `totalEdge` 的所有 1-5 边的序号为 3.

- 只要把 `totalJ` 恢复成三列即得所有三角形单元边的自然序号, 这是因为 `totalEdge` 排列的规则是: 前 NT 行对应所有单元的第 1 条边, 中间 NT 行对应第 2 条边, 最后 NT 行对应第 3 条边. 综上, 可如下获取 `elem2edge`.

```

1 [node, elem] = squaremesh([0 1 0 1], 0.5);
2 totalEdge = sort([elem(:, [2, 3]); elem(:, [3, 1]); elem(:, [1, 2])], 2);
3 [edge, i1, totalJ] = unique(totalEdge, 'rows');
4 NT = size(elem, 1);
5 elem2edge = reshape(totalJ, NT, 3);

```

结果如下

	1	2	3
1	3	1	5
2	6	4	7
3	10	8	12
4	13	11	14
5	3	8	2
6	6	11	5
7	10	15	9
8	13	16	12

edge2elem 的生成

对给定的一条边 e , 有时候希望知道包含它的单元有哪些. 对内部边, 就是哪两个单元以 e 为公共边. 为此定义矩阵 `edge2elem`, 其维数为 $NE \times 2$, 而 NE 是一维边的个数. 它的第一列为左单元编号, 第二列为右单元编号. 注意, 对边界边规定两个编号一致.

	1	2
1	1	5
2	2	6
3	4	8
4	5	9
5	1	5
6	2	6
7	4	8
8	5	9
9	1	2
10	2	3
11	4	5
12	5	6
13	4	5
14	5	6
15	7	8
16	8	9
17	2	5
18	3	6
19	5	8
20	6	9
21	1	4
22	2	5
23	4	7
24	5	8

(a) totalEdge

	1	2
1	1	2
2	1	4
3	1	5
4	2	3
5	2	5
6	2	6
7	3	6
8	4	5
9	4	7
10	4	8
11	5	6
12	5	8
13	5	9
14	6	9
15	7	8
16	8	9
17	2	5
18	3	6
19	5	8
20	6	9
21	1	4
22	2	5
23	4	7
24	5	8

(b) edge

	1
1	9
2	21
3	1
4	10
5	17
6	2
7	18
8	11
9	23
10	3
11	12
12	19
13	8
14	11
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

(c) i1

	1
1	3
2	6
3	10
4	13
5	3
6	6
7	10
8	13
9	1
10	4
11	8
12	11
13	8
14	11
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

(d) totalJ

- `totalEdge` 记录了所有的重复边, 称第一次出现的重复边为左单元边, 第二次出现的重复边为右单元边. 根据前面的说明,

```
[~, i1, totalJ] = unique(totalEdge, 'rows');
```

执行上面语句给出的 `i1` 记录了左单元边在 `totalEdge` 中的行号.

- 类似地, 对 `totalEdge` 的逆序使用 `unique`:

```
[~, i2] = unique(totalEdge(end:-1:1,:), 'rows');
```

或

```
[~, i2] = unique(totalJ(end:-1:1), 'rows');
```

给出的 i_2 记录了右单元边在 $totalEdge$ 中的行号, 但现在的序号与原先的有差别. 以图中的例子为例, 此时 1 相当于原来的 24, 2 相当于 23, 依此类推. 它们的和总是 25, 即 $\text{length}(totalEdge)+1$ (三角形为 $3 \times NT+1$). 这样, 还原后的为

```
i2 = length(totalEdge)+1-i2;
```

- 其实可以直接用 i_1 和 $totalJ$ 获得 i_2 .

```
i2(totalJ)= 1:length(totalJ); i2 = i2(:);
```

这是简单的覆盖技巧.

- 例如考察图 2.2 中 $totalJ$ 的第 1 行和第 5 行, 它们都对应 $edge$ 中的第 3 条边.
- 上面获得 $i_2(3)$ 的过程为

```
i2(3)= 1; i2(3)= 5;
```

- 即根据行索引与 3 的对应, 把第一次出现的索引 1 用第二次出现的索引 5 覆盖.

- $totalJ$ 或 $totalEdge$ 并不是逐个单元存储的. 对三角剖分, 它是如下存储的

所有单元的第 1 条边: `elem(:, [2,3]); % NT * 2`

所有单元的第 2 条边: `elem(:, [3,1]); % NT * 2`

所有单元的第 3 条边: `elem(:, [1,2]); % NT * 2`

即, 前 NT 行与所有单元的第 1 条边对应, 中间的 NT 行与所有单元的第 2 条边对应, 最后的 NT 行与所有单元的第 3 条边对应. 设 $totalJ$ 行的单元序号为 $totalJ_{elem}$, 则

```
totalJ_{elem} = repmat((1:NT)', 3, 1);
```

- 对三角形剖分, 有

```
edge2elem = totalJ_{elem}([i1, i2]);
```

综上, 如下实现 $edge2elem$

```
1 totalJ_{elem} = repmat((1:NT)', 3, 1);
2 [~, i2] = unique(totalJ(end:-1:1), 'rows');
3 i2 = length(totalEdge)+1-i2;
4 edge2elem = totalJ_{elem}([i1, i2]);
```

注 2.1 $totalEdge$ 是按单元顺序排列的, i_1 对应 e 第一次出现的单元, i_2 对应第二次出现的单元, 自然 $edge2elem$ 的第一列单元序号小于或等于第二列单元序号.

在某些计算中, 可能希望知道给定边在左右单元中的局部序号. 为此, 以下将 `edge2elem` 增加两列, 其中, 第 3 列存储边在左单元中的局部序号, 第 4 列存储边在右单元中的局部序号.

- 获得局部序号的关键是 `totalEdge` 的排列规则, 即前 NT 行与所有单元的第 1 条边对应, 中间的 NT 行与所有单元的第 2 条边对应, 最后的 NT 行与所有单元的第 3 条边对应.

所有单元的第 1 条边: `elem(:, [2,3]); % NT * 2`

所有单元的第 2 条边: `elem(:, [3,1]); % NT * 2`

所有单元的第 3 条边: `elem(:, [1,2]); % NT * 2`

这样, 给定边落在前 NT 行, 中间 NT 行和最后 NT 行的局部序号分别为 1, 2 和 3.

- `i1` 记录了左单元边在 `totalEdge` 中的行号, 对应的局部序号为 `ceil(i1/NT)`. 注意,

`ceil(1/3) = 1, ceil(2/3) = 1, ceil(3/3) = 1`

`ceil(4/3) = 2, ceil(5/3) = 2, ceil(6/3) = 2`

即向右取整, 也就是取大于或等于真实值的最小整数.

- 新的 `edge2elem` 为

```
edge2elem = [edge2elem, ceil(i1/NT), ceil(i2/NT)];
```

neighbor 的生成

`neighbor` 与 `elem2edge` 类似, 只不过记录的是每个单元各条边连接的单元序号 (对边界单元, 规定其边界边连接的单元为自身).

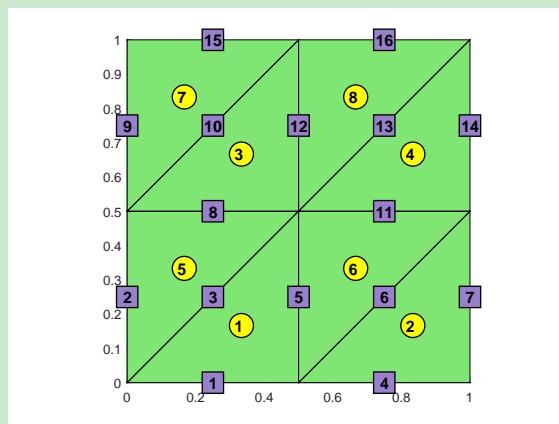


图 2.3. 三角剖分边的自然序号

- 以图 2.3 的单元 3 为例, 它的边的局部顺序为 10-8-12. 这些边对应的左右单元如下 (左右顺序其实无所谓, 但前面已经保证左侧序号小于右侧序号)

$$\begin{bmatrix} \text{左: } & 3 & 3 & 3 \\ \text{右: } & 7 & 5 & 8 \end{bmatrix}$$

- 将该矩阵按列拉直, 并去除当前单元的序号 3, 所得向量 [7, 5, 8] 即为相邻单元的序号. 但对边界单元, 这种处理会丢失边界边. 例如, 单元 1 边的局部顺序为 3-1-5, 每条边对应的左右单元为

$$\begin{bmatrix} \text{左: } & 1 & 1 & 1 \\ \text{右: } & 5 & 1 & 6 \end{bmatrix}$$

去除重复后为 [5, 6], 边界边没有保留序号.

我们换一种处理方法, 先给出程序.

```

1 % ----- neighbor -----
2 neighbor = cell(NT,1);
3 for iel = 1:NT
4     index = elem2edge{iel};
5     ia = edge2elem(index,1); ib = edge2elem(index,2);
6     ia(ia==iel) = ib(ia==iel);
7     neighbor{iel} = ia;
8 end

```

- ia 和 ib 分别是左侧和右侧单元的序号.
- 左侧单元序号 ia 中有一部分是当前单元序号 iel, 它们应该替换成对应的右侧单元序号 ib(ia==iel).
- 上面的替换保留了矩阵

$$\begin{bmatrix} \text{左: } & 1 & 1 & 1 \\ \text{右: } & 5 & 1 & 6 \end{bmatrix}$$

中第 2 列的左侧单元序号 1, 因而保留了边界边对应的单元.

2.2 网格相关的几何量

几何量包括

表 2.2. 几何量

centroid	单元重心坐标
area	单元面积
diameter	单元直径

- 单元的重心如下计算

$$x_K = \frac{1}{6|K|} \sum_{i=0}^{N_v-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$y_K = \frac{1}{6|K|} \sum_{i=0}^{N_v-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

这里 N_v 是单元顶点个数.

- 单元面积

$$|K| = \frac{1}{2} \left| \sum_{i=0}^{N_v-1} x_i y_{i+1} - x_{i+1} y_i \right|.$$

- 单元直径就是所有顶点之间最长的距离, MATLAB 提供了 pdist 函数, 它计算各对行向量的相互距离.

以上几何量可如下获得

```

1 % ----- centroid, area, diameter -----
2 centroid = zeros(NT,2); area = zeros(NT,1); diameter = zeros(NT,1);
3 s = 1;
4 for iel = 1:NT
5     index = elem(iel,:);
6     verts = node(index, :); verts1 = verts([2:end,1],:);
7     area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);
8     ar = 0.5*abs(sum(area_components));
9     area(iel) = ar;
10    centroid(s,:) = sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);
11    diameter(s) = max(pdist(verts));
12    s = s+1;
13 end

```

2.3 auxstructure 与 auxgeometry 函数

为了输出方便, 我们把所有的数据结构或几何量保存在结构体 aux 中. 考虑到数据结构在编程中不一定使用 (处理网格时用), 我们把数据结构与几何量分别用函数生成, 命名为 auxstructure.m 和 auxgeometry.m. 为了方便使用, 程序中把三角剖分按单元存储的数据转化为元胞数组. auxstructure.m 函数如下

CODE 2.1. auxstructure.m

```

1 function aux = auxstructure(node,elem)
2
3 NT = size(elem,1);
4

```

```

5 %% elem2edge
6 totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
7 [~, i1, totalJ] = unique(totalEdge, 'rows');
8 elem2edge = reshape(totalJ,NT,3);
9
10 %% edge, bdEdge
11 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
12 edge = [j,i];
13 bdEdge = edge(s==1,:); % not counterclockwise
14
15 %% edge2elem
16 totalJelem = repmat((1:NT)',3,1);
17 [~, i2] = unique(totalJ(end:-1:1), 'rows');
18 i2 = length(totalEdge)+1-i2;
19 edge2elem = totalJelem([i1,i2]);
20 edge2elem = [edge2elem, ceil(i1/NT), ceil(i2/NT)];
21
22 %% neighbor
23 neighbor = zeros(NT,dim);
24 for iel = 1:NT
25     indexE = elem2edge(iel,:);
26     ia = edge2elem(indexE,1); ib = edge2elem(indexE,2);
27     ia(ia==iel) = ib(ia==iel);
28     neighbor(iel,:) = ia';
29 end
30
31
32 aux.node = node; aux.elem = elem;
33 aux.elem2edge = elem2edge;
34 aux.edge = edge; aux.bdEdge = bdEdge;
35 aux.edge2elem = edge2elem;
36 aux.neighbor = neighbor;

```

auxgeometry.m 函数如下

CODE 2.2. auxgeometry.m

```

1 function aux = auxgeometry(node,elem)
2
3 % ----- centroid, area, diameter -----
4 NT = size(elem,1);
5 centroid = zeros(NT,2); area = zeros(NT,1); diameter = zeros(NT,1);
6 s = 1;
7 for iel = 1:NT
8     index = elem(iel,:);
9     verts = node(index, :); verts1 = verts([2:end,1],:);
10    area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);
11    ar = 0.5*abs(sum(area_components));
12    area(iel) = ar;
13    Centroid(s,:) = sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);

```

```

14     diameter(s) = max(pdist(verts));
15     s = s+1;
16 end
17
18 aux.node = node; aux.elem = elem;
19 aux.centroid = centroid;
20 aux.area = area;
21 aux.diameter = diameter;

```

2.4 边界设置

假设网格的边界只有 Dirichlet 与 Neumann 两种类型, 前者用 `eD` 存储 Dirichlet 节点的编号, 后者用 `elemN` 存储 Neumann 边界的起点和终点编号 (即一维问题的连通性信息). 为了方便, 有时候需要 Dirichlet 边的信息, 为此我们用 `elemD` 存储 Dirichlet 边界的起点和终点编号.

2.4.1 边界边的定向

辅助数据结构中曾给出了边界边 `bdEdge`, 但它的定向不再是逆时针, 因为我们规定 $\text{edge}(k, 1) < \text{edge}(k, 2)$. Neumann 边界条件中会遇到 $\partial_n u$, 这就需要我们恢复边界边的定向以确定外法向量 (边的旋转获得).

给定 `totalEdge`, 即所有单元的边 (含重复且无定向), 我们有两种方式获得边 (第一种可获得边界边, 第二种只获得所有边):

- 一是累计重复的次数 (1 是边界, 2 是内部)

```

1 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
2 edge = [j,i];
3 bdEdge = edge(s==1,:);

```

- 二是直接去掉重复的边

```

1 [edge, i1, ~] = unique(totalEdge, 'rows');

```

这里, `i1` 记录的是 `edge` 在重复边 `totalEdge` 中的位置.

显然, `i1(s==1)` 给出的是边界边 `bdEdge` 在 `totalEdge` 中的位置. `totalEdge` 的原来定向是知道的 (它由 `allEdge` 可知), 由此就可确定边界边的定向, 程序如下

```

1 [node,elem] = squaremesh([0 1 0 1],0.5);
2 NT = size(elem,1);
3 % totalEdge
4 if iscell(elem)

```

```

5      shiftfun = @(verts) [verts(2:end),verts(1)];
6      T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
7      v0 = horzcat(elem{:})'; % the starting points of edges
8      v1 = horzcat(T1{:})'; % the ending points of edges
9      allEdge = [v0,v1];
10     totalEdge = sort(allEdge,2);
11 else
12     allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
13     totalEdge = sort(allEdge,2);
14 end
15 [~,~,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
16 [edge, i1, ~] = unique(totalEdge, 'rows');
17 bdEdge = allEdge(i1(s==1),:); % counterclockwise

```

注 2.2 边界边 `bdEdge` 在一维边集合 `edge` 中的自然序号为

```
bdIndex = find(s==1);
```

2.4.2 边界的设置

下面说明如何实现 `eD`, `elemD` 和 `elemN`. 以下图为例

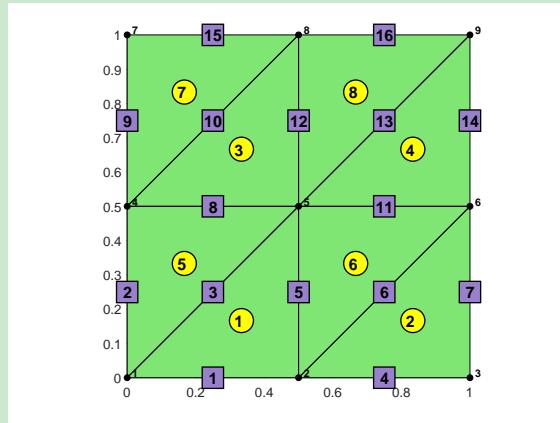


图 2.4. 边的自然序号

- 边界边的序号顺序为 $1, 2, 4, 7, 9, 14, 15, 16$. 定向的 `bdEdge` 给出的是这些边的起点与终点编号, 只要按索引对应即可.
- 边界我们用函数确定, 例如矩形 $[0,1]^2$ 的右边界为满足 $x = 1$ 的线段组成. 只需要判断 `bdEdge` 对应的边的中点在不在该线段上. 如下

```

1 bdFun = 'x==1';
2 nodebdEdge = (node(bdEdge(:,1),:) + node(bdEdge(:,2),:))/2;
3 x = nodebdEdge(:,1); y = nodebdEdge(:,2);
4 id = eval(bdFun);

```

这里, `eval` 将字符串视为语句并运行. 现在给定了若干个 `x`, 执行 `eval(bdFun)` 就会判断哪些 `x` 满足条件, 返回的是逻辑数组 `id = [0 0 0 1 0 1 0 0]'`, 即索引中的第 4,6 条边在右边界上.

- 这样, 我们就可抽取出需要的边 `bdEdge(id,:)`. 需要注意的是, `node` 在边界上不一定精确为 1, 通常将上面的 `bdFun` 修改为

```
bdFun = 'abs(x-1)<1e-4';
```

- Neumann 边界通常比 Dirichlet 边界少, 为此在建函数的时候, 输入的字符串默认认为是 Neumann 边界的, 其他的都是 Dirichlet. 另外, 当没有边界字符串的时候, 规定所有边都是 Dirichlet 边.

根据上面的讨论, 我们可以给出函数 `setboundary.m`

CODE 2.3. `setboundary.m`

```

1 function bdStruct= setboundary(node,elem,varargin)
2 % varargin: string for Neumann boundary
3
4 % ----- totalEdge -----
5 if iscell(elem)
6     shiftfun = @(verts) [verts(2:end),verts(1)]; % or shiftfun = @(verts) ...
    circshift(verts,-1);
7     T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
8     v0 = horzcat(elem{:})'; % the starting points of edges
9     v1 = horzcat(T1{:})'; % the ending points of edges
10    allEdge = [v0,v1];
11 else % Triangulation
12    allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
13 end
14 totalEdge = sort(allEdge,2);
15
16 % ----- counterclockwise bdEdge -----
17 [~,~,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
18 [~, i1, ~] = unique(totalEdge,'rows');
19 bdEdge = allEdge(i1(s==1),:);
20
21 % ----- set boundary -----
22 nE = size(bdEdge,1);
23 % initial as Dirichlet (true for Dirichlet, false for Neumann)
24 bdFlag = true(nE,1);
25 nodebdEdge = (node(bdEdge(:,1),:) + node(bdEdge(:,2),:))/2;
26 x = nodebdEdge(:,1); y = nodebdEdge(:,2);
27 nvar = length(varargin); % 1 * size(varargin,2)
28 % note that length(varargin) = 1 for bdNeumann = [] or ''
29 if (nargin==2) || (~isempty(varargin{1}))
```

```

30     for i = 1:nvar
31         bdNeumann = varargin{i};
32         id = eval(bdNeumann);
33         bdFlag(id) = false;
34     end
35 end
36 bdStruct.eD = unique(bdEdge(bdFlag,:));
37 bdStruct.elemN = bdEdge(~bdFlag,:);

```

这里, ed 和 elemN 保存在结构体 bdStruct 中. 例如,

1. 以下命令给出的边界全是 Dirichlet 边界:

```

bdStruct = setboundary(node,elem);
bdStruct = setboundary(node,elem, []);
bdStruct = setboundary(node,elem, '');

```

2. bdStruct = setboundary(node,elem,'x==1') 将右边界设为 Neumann 边, 其他为 Dirichlet 边.
3. bdStruct = setboundary(node,elem,'(x==1)|(y==1)') 将右边界与上边界设为 Neumann 边.

注 2.3 以下两种写法等价

```

bdStruct = setboundary(node,elem,'(x==1)|(y==1)');
bdStruct = setboundary(node,elem,'x==1','y==1');

```

第三章 三角网格的生成

本文考虑 DistMesh (A Simple Mesh Generator in MATLAB) 工具箱, 如下网页给出了下载链接及简单的介绍

<http://persson.berkeley.edu/distmesh/>

3.1 Delaunay 三角剖分

3.1.1 Delaunay 三角剖分的定义

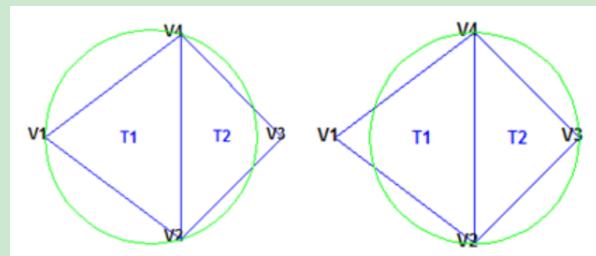
定义 3.1 (三角剖分) 假设 V 是二维实数域上的有限点集, 边 e 是由点集中的点作为端点构成的封闭线段, E 为 e 的集合. 称平面图 $\mathcal{T} = (V, E)$ 是点集 V 的三角剖分, 如果它满足条件:

1. 除了端点, 平面图中的边不包含点集中的任何点, 即边与点集无交点 (除端点);
2. 没有相交边, 即边和边没有交点 (除端点);
3. 所有的面都是三角面, 且所有三角面的合集是点集 V 的凸包.

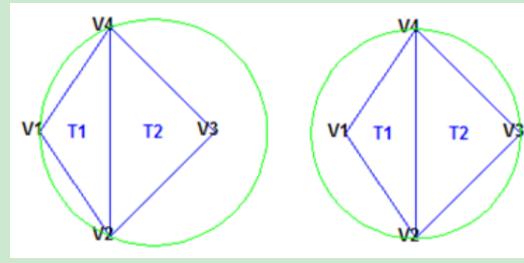
实际应用中, 最常用的是 Delaunay 三角剖分, 它是一种特殊的三角剖分. 为此, 我们先给出 Delaunay 边的定义.

定义 3.2 (Delaunay 边) 任给边集合 E 中的一条边 e , 设其两个端点为 a, b . 称 e 为 Delaunay 边, 如果它满足空圆性质: 存在一个圆经过 a, b 两点, 且该圆内不含点集 V 中任何其他的点.

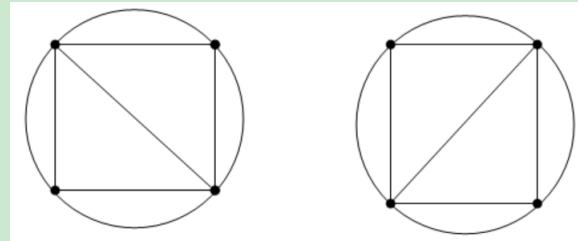
定义 3.3 (Delaunay 三角剖分) 点集 V 的一个三角剖分 \mathcal{T} 称为 Delaunay 三角剖分, 如果 \mathcal{T} 的每条边都是 Delaunay 边.



如图, 设 $e = \overline{v_2v_4}$ 是三角剖分中的一条边, 以该边为公共边的三角形为 T_1 和 T_2 . 可以证明, 若 T_1 的外接圆不包含 v_3 , 且 T_2 的外接圆不包含 v_1 , 则 e 就是 Delaunay 边. 下图中的 $e = \overline{v_2v_4}$ 就不是 Delaunay 边.



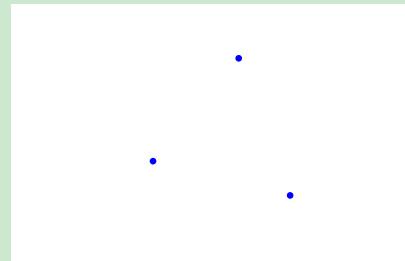
一般而言, Delaunay 三角剖分是不唯一的, 这是因为可能存在四点共圆, 如下图所示.



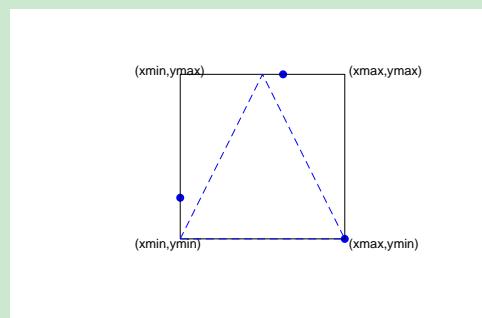
3.1.2 Bowyer 逐点插入法

Step 1: 构造超级三角形以包含所有点

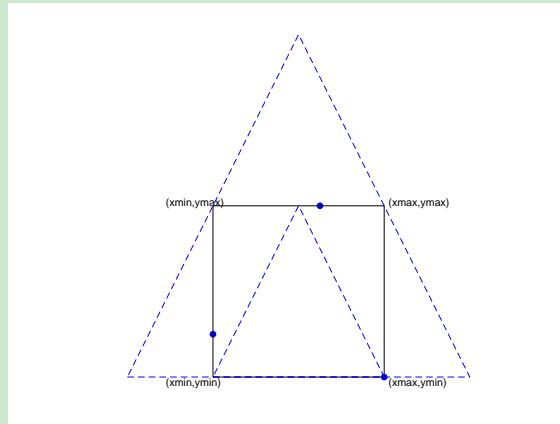
以如下的三个点进行说明.



先用坐标范围大小的矩形包含所有点, 并给定矩形内部的一个三角形, 如下图.



这里三角形的顶点为上侧边的中点. 做该三角形的相似三角形, 使其过矩形的两个上方顶点.



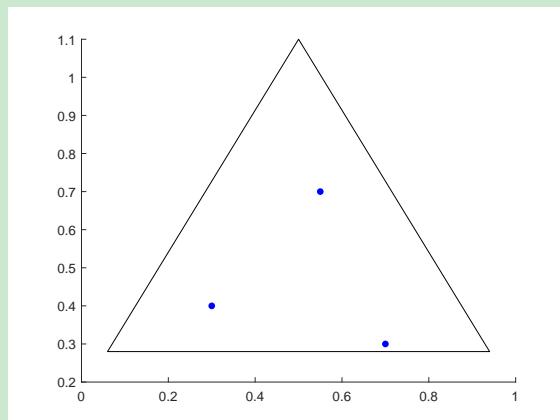
接着, 我们把三角形的底边往下移动一点并适当延长, 所得三角形一定包含所有散点 (都是内点).

```

1 % ----- vertex list -----
2 BdBox = [0 1 0 1]; h0 = 0.5;
3 [x,y] = meshgrid(BdBox(1):h0:BdBox(2),BdBox(3):h0*sqrt(3)/2:BdBox(4));
4 p = [x(:,1),y(:,1)];
5 np = size(p,1);
6
7 % ----- determine the supertriangle -----
8 xmin = min(p(:,1)); xmax = max(p(:,1));
9 ymin = min(p(:,2)); ymax = max(p(:,2));
10 hx = xmax-xmin; hy = ymax-ymin;
11
12 eps = 0.05*hy;
13 pts = [
14     xmin+hx/2,           ymax+hy;
15     xmin-hx/2-2*eps,   ymin-eps;
16     xmax+hx/2+2*eps,   ymin-eps
17 ];
18 tri = [1 2 3];

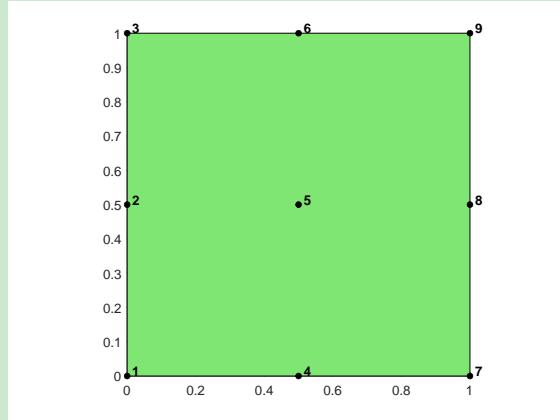
```

这里用 `pts` 和 `tri` 分别存储生成的三角形的顶点坐标和连通性.



注 3.1 超级三角形的顶点和预先给定的散点就是三角剖分的顶点, 因此最终的节点以及初始的连通性信息为

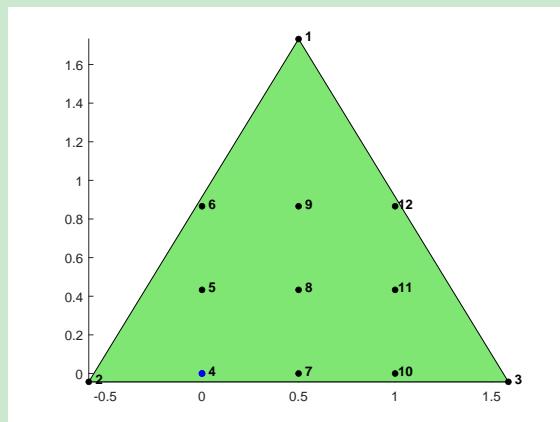
```
1 node = [pts; p];
2 elem = [1 2 3];
```



为了方便, 考虑上图中的 9 个散点. 注意到要加上超级三角形的三个顶点, 上面的顶点序号也要加上 3.

Step 2: 添加第 1 个点

当前数据结构如下图



我们要找到第 1 个散点 (即图中的第 4 个点) 的影响三角形, 也就是外接圆包含该点的三角形. 函数 `inCircle(p0, p1, p2, p3)` 判断点 p_0 是否在以 p_1, p_2, p_3 为顶点的三角形的外接圆内. 这样, 可如下找到所有影响三角形.

```
1 id_tri = false(NT,1); % extracting index
2 for iel = 1:NT
3     index = elem(iel,:);
4     z1 = node(elem(iel,1),:);
```

```

5      z2 = node(elem(iel,2),:);
6      z3 = node(elem(iel,3),:);
7      if inCircle(pp,z1,z2,z3), id_tri(iel) = true; end
8 end
9 tri = elem(id_tri,:);

```

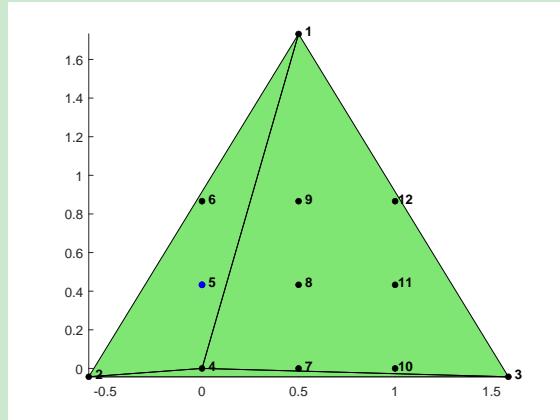
所有影响三角形构成一个 n 边形, 上面的情形只有一个影响三角形(后面添加点时可看到多个三角形的情形). 连接散点和多边形的顶点得到当前的三角剖分.

```

1 tri = elem(id_tri,:);
2 edge_buffer = [tri(:,[2,3]); tri(:,[3,1]); tri(:,[1,2])];
3 N = 3+k; neb = size(edge_buffer,1);
4 tri = [edge_buffer, N*ones(neb,1)]; % new triangles
5 elem = [elem(~id_tri,:); tri];

```

最后一步是把非影响三角形和新产生的三角形合并得到当前的三角剖分.



Step 3: 添加第 2 个点

现在添加第 2 个散点, 即图中的点 5. 类似前面可找到影响三角形为 3-1-4 和 1-2-4, 即上图的左右两个三角形. 这两个三角形对应的多边形为 3-1-2-4, 显然只需要记录两个三角形的非公共边. 为此, 我们要删除 `edge_buffer` 中的所有重复边.

```

1 aa = full(sparse(edge_buffer(:,1),edge_buffer(:,2),1));
2 aa(aa+aa'>=2) = 0;
3 [i,j] = find(aa);
4 edge_poly = [i,j];

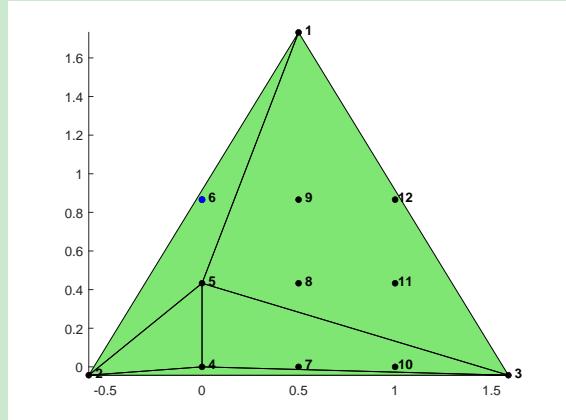
```

连接多边形的顶点与当前散点即得此时的三角剖分.

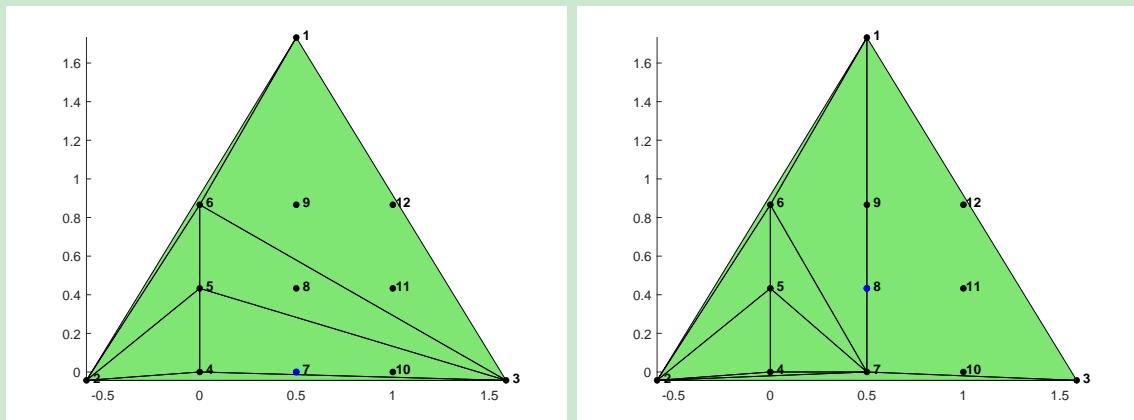
```

1 N = 3+k; n_poly = size(edge_poly,1);
2 tri_poly = [edge_poly, N*ones(n_poly,1)]; % new triangles
3 elem = [elem(~id_tri,:); tri_poly];

```

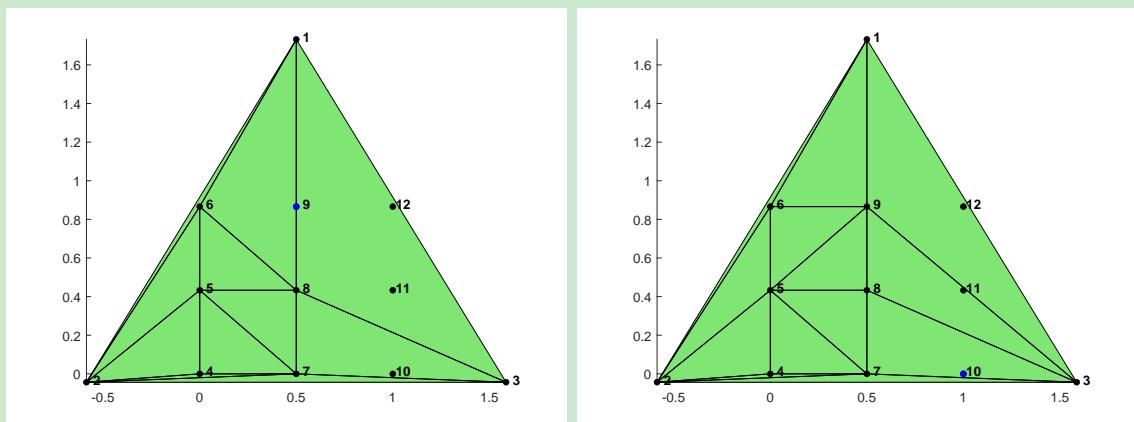


依次添加散点所得三角剖分图分别为



(a) $k = 3$

(b) $k = 4$



(c) $k = 5$

(d) $k = 6$

可以看到, $k = 6$ 时出现矩形对角线变方向的情形 (本程序不打算更改方向, 因为它一般实现的是无结构网格).

Step 4: 去掉超级三角形相关的三角形

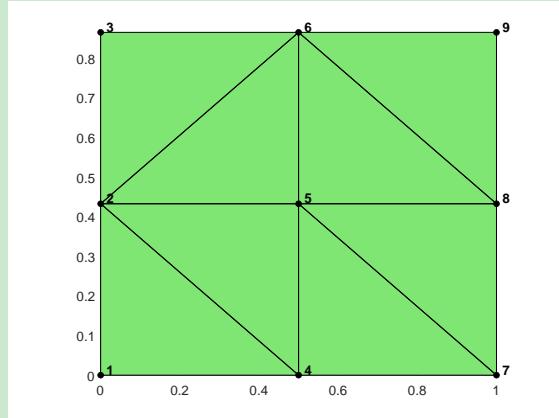
按照前面的过程遍历所有散点后, 我们只要去掉超级三角形顶点相关的三角形即得最终的三角剖分.

```

1 % remove any triangles that use the supertriangle vertices
2 NT = size(elem,1);
3 id = (elem(:,1)>3) & (elem(:,2)>3) & (elem(:,3)>3);
4 t = elem(id,:)-3;

```

剖分图如下



注 3.2 在 inCircle.m 中, 采用

[Wikipedia:https://en.wikipedia.org/wiki/Delaunay_triangulation](https://en.wikipedia.org/wiki/Delaunay_triangulation)

中给出的行列式进行判断. 点在外接圆内, 对应行列式大于零. 但从计算实践可以发现, 最好采用

```

1 id = detA>itol; 或 id = detA>-itol;

```

代替, 程序中取前者, 且令 $itol = 1e-6$.

3.1.3 程序整理

最终的程序如下

```

1 function t = my_delaunay(p)
2
3 % determine the supertriangle
4 xmin = min(p(:,1)); xmax = max(p(:,1));
5 ymin = min(p(:,2)); ymax = max(p(:,2));
6 hx = xmax-xmin; hy = ymax-ymin;
7
8 eps = 0.05*hy;
9 pts =
10    xmin+hx/2,           ymax+hy;
11    xmin-hx/2-2*eps,   ymin-eps;
12    xmax+hx/2+2*eps,   ymin-eps
13];
14 node = [pts; p];    elem = [1 2 3];
15

```

```

16 for k = 1:size(p,1)
17     % find effected triangles assocaited with k-th point
18     z1 = node(elem(:,1,:));
19     z2 = node(elem(:,2,:));
20     z3 = node(elem(:,3,:));
21     id_tri = inCircle(p(k,:),z1,z2,z3);
22     tri = elem(id_tri,:); % effected triangles
23     edge_buffer = [tri(:,[2,3]); tri(:,[3,1]); tri(:,[1,2])];
24
25     % obtain edges of the enclosing polygon
26     aa = full(sparse(edge_buffer(:,1),edge_buffer(:,2),1));
27     aa(aa+aa'>=2) = 0;
28     [i,j] = find(aa);
29     edge_poly = [i,j];
30
31     % update elem
32     N = 3+k; n_poly = size(edge_poly,1);
33     tri_poly = [edge_poly, N*ones(n_poly,1)]; % new triangles
34     elem = [elem(~id_tri,:); tri_poly];
35 end
36
37 % remove any triangles that use the supertriangle vertices
38 id = (elem(:,1)>3) & (elem(:,2)>3) & (elem(:,3)>3);
39 t = elem(id,:)-3;
40 end % end of my_delaunay

```

注意, 我们修改了 inCircle.m 函数, 它可以判断一个点与多个三角形外接圆的位置关系.

注 3.3 相比 MATLAB 内置函数 delaunay.m 或 delaunayn.m, 本文介绍的算法要慢很多 (对过细的网格, 有可能出错).

3.2 DistMesh

3.2.1 DistMesh 的网格优化思想

给定区域中的一些散点, Delaunay 可生成这些散点对应的三角剖分. 显然, 网格的质量与散点的分布有关系. 另外, Delaunay 三角剖分不能直接解决区域边界的近似. 本文将介绍 Persson 和 Strang 在文 [7] 中提出的网格优化算法, 相应的 MATLAB 工具包为 DistMesh, 可在如下网页获取

<http://persson.berkeley.edu/distmesh/>

DistMesh 的算法可视为一种力学方法 (a force-based method), 它把三角网格视为弹性力学中的桁架, 其中的边对应桁架的杆件, 散点对应杆件连接的节点. 假设桁架上有某种力-位移关系, DistMesh 将平衡位置的节点作为最终的 Delaunay 三角剖分的散点.

力 \mathbf{F} 是点 \mathbf{p} 的函数, 即 $\mathbf{F} = \mathbf{F}(\mathbf{p})$ (该点处的合力). 注意它是向量, 有水平和垂直方向 (即 x 和 y 方向) 的两个分量, 记为 $\mathbf{F} = [\mathbf{F}_x, \mathbf{F}_y]$. 它的大小记为 $f(\ell, \ell_0)$, 即杆件上的力-位移关系函数, 仅依赖于杆件当前的长度 ℓ 和平衡时的期望长度 ℓ_0 .

力-位移关系有多种选择, DistMesh 中考虑如下简单的线性关系

$$f(\ell, \ell_0) = \begin{cases} k(\ell_0 - \ell), & \ell < \ell_0, \\ 0, & \ell \geq \ell_0, \end{cases}$$

为了方便, 程序中取 $k = 1$. 注意它并不是 Hooke 定律, 因为杆件并不一定是沿着杆方向在移动. DistMesh 把初始给定的三角拓扑结构想象成一个压缩的桁架结构 (杆可理解为弹簧). 该结构往外扩展时, 弹簧上的力是排斥力. 当弹簧达到期望长度 ℓ_0 后, 它不需要再扩展, 因而设其力为零 (注意此时其他弹簧仍会拉扯该弹簧, 导致其伸长, 但仍不考虑这个拉长后的吸引力).

我们将采用某种迭代法获得最终的散点, 在接近期望长度 ℓ_0 时, 自然也要求大部分杆件上 $f > 0$. 为此可取 ℓ_0 比实际迭代长度稍大, 如 $\ell_0 = 1.2\ell$ (`Fscale = 1.2`), 这里的 ℓ 是迭代过程中杆的“平均长度”.

3.2.2 符号距离函数

定义 3.4 设 $\Omega \subset \mathbb{R}^2$, 定义 $d_\Omega : \mathbb{R}^2 \rightarrow \mathbb{R}$ 为

$$d_\Omega(x) = s_\Omega(x) \min_{y \in \partial\Omega} |x - y|,$$

其中,

$$s_\Omega(x) = \begin{cases} -1, & x \in \Omega, \\ +1, & x \in \mathbb{R}^2 \setminus \Omega. \end{cases}$$

称 d_Ω 为符号距离函数, 获得最小值的边界点称为关于 x 的最近边界点.

显然有

$$\overline{\Omega} = \{x \in \mathbb{R}^2 : d_\Omega(x) \leq 0\}, \quad \partial\Omega = \{x \in \mathbb{R}^2 : d_\Omega(x) = 0\}.$$

对圆心在 x_0 , 半径为 r 的球, 易知

$$d_\Omega(x) = |x - x_0| - r.$$

当 Ω 光滑时, $\nabla d_\Omega(x)$ 是最近边界点处的单位法向量, 例如对圆, 有 $\nabla d_\Omega(x) = \text{sgn}(x - x_0)$. 一般地, 对几乎所有的 $x \in \mathbb{R}^2$, 有 $|\nabla d_\Omega(x)| = 1$.

通过图形观察可知, x 关于最近边界点的反射点为

$$R_\Omega(x) = x - 2d_\Omega(x)\nabla d_\Omega(x). \quad (3.1)$$

对复合区域, 通常有

$$d_{\Omega_1 \cup \Omega_2}(\mathbf{x}) = \min(d_{\Omega_1}(\mathbf{x}), d_{\Omega_2}(\mathbf{x})),$$

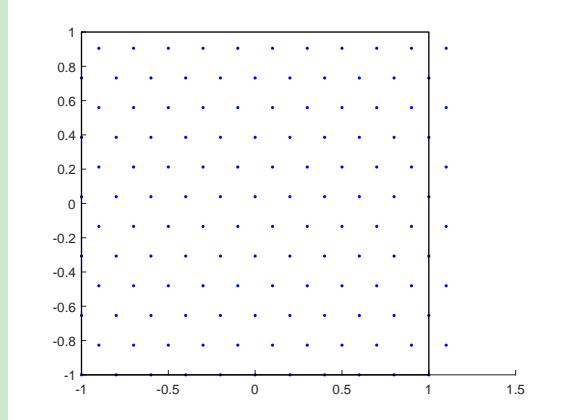
$$d_{\Omega_1 \cap \Omega_2}(\mathbf{x}) = \max(d_{\Omega_1}(\mathbf{x}), d_{\Omega_2}(\mathbf{x})),$$

$$d_{\mathbb{R}^2 \setminus \Omega_1}(\mathbf{x}) = -d_{\Omega_1}(\mathbf{x}).$$

3.2.3 程序说明

初始网格点的生成

对给定的区域, 设 $\text{BdBox} = [\text{xmin } \text{xmax } \text{ymin } \text{ymax}]$ 是覆盖它的矩形框. 在该矩形区域中, 生成等边三角形分布的节点. 做法是首先生成矩形网格, 然后平移偶数层, 如下图所示



```

1 % 1. Create initial distribution in bounding box (equilateral triangles)
2 x = BdBox(1):h0:BdBox(2); y = BdBox(3):h0*sqrt(3)/2:BdBox(4);
3 [x,y] = meshgrid(x,y);
4 x(2:2:end,:) = x(2:2:end,:)+h0/2;                                % Shift even rows
5 p = [x(:),y(:)];                                                 % List of node coordinates

```

这里的 h_0 是正三角形的边长, 它的高为 $\sqrt{3}/2h_0$.

接着要移除区域外的点

```

1 p = p(fd(p)<geps,:);                                     % Keep only d<0 points

```

这里的 fd 是区域的符号距离函数, 而 $d < 0$ 中的 0 用一个小参数 $geps$ 代替. 对某些区域, 希望初始点中含有一些固定的节点, 如矩形区域四个角点.

```

1 p = setdiff(p,pfix,'rows');
2 p = [pfix; p];
3 N = size(p,1);

```

在做 Delaunay 三角剖分时, 一般节点是不同的, 上面去掉固定点导致的重复 (setdiff 是集合运算中的差运算), 即删除 p 中与 $pfix$ 重复的点 (注意一般 $pfix$ 不重复, 它是人为给定的).

Delaunay 三角剖分

算法是对初始散点进行迭代, 找到“最佳”的位置分布. 对给定的散点, 要建立桁架结构, 即 Delaunay 三角剖分, 如下

```

1 % 3. Get the truss and bars by the Delaunay algorithm
2 t = delaunay(p);
3 pm = (p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3; % Compute centroids
4 t = t(fd(pm)<-geps,:); % Keep interior triangles
5 totalEdge = sort([t(:,[2 3]); t(:,[3 1]); t(:,[1 2])],2);
6 bars = unique(totalEdge,'rows'); % Bars as node pairs

```

这里要删除区域外的三角形, 为了方便, 仅删除重心在区域外的点, 而 bars 就是网格中的边.

节点迭代

接着, 要根据桁架结构, 用力学方法进行迭代. 平衡位置的 \mathbf{p} 对应 $\mathbf{F}(\mathbf{p}) = \mathbf{0}$. 该问题不太容易求解, 一个简单的替代方法是引入时间依赖, 即求解如下的 ODEs

$$\begin{cases} \frac{d\mathbf{p}}{dt} = \mathbf{F}(\mathbf{p}), & t \geq 0, \\ \mathbf{p}(0) = \mathbf{p}_0. \end{cases}$$

这是因为它的稳态解满足 $\mathbf{F}(\mathbf{p}) = \mathbf{0}$. 用简单的前向 Euler 公式求解

$$\mathbf{p}_{n+1} = \mathbf{p}_n + \Delta t \mathbf{F}(\mathbf{p}_n).$$

注意上面的 \mathbf{F} 是点 \mathbf{p}_n 处的合力.

杆在节点处力的大小为 f , 由此可得每个杆上的力向量 \mathbf{F} .

```

1 barvec = p(bars(:,2),:)-p(bars(:,1),:); % List of bar vectors
2 L = sqrt(sum(barvec.^2,2)); % Bar lengths
3 L0 = Fscale*sqrt(sum(L.^2)/length(L)); % Desired lengths
4 f = max(L0-L,0); % Bar forces (scalars)
5 F = f./L*[1,1].*barvec; % Bar forces (x,y components)

```

这里 $F_{\text{scale}} = 1.2$, 平均长度取为

$$\sqrt{\frac{L_1^2 + \cdots + L_n^2}{n}}.$$

注意, barvec 是从 bar 的起点指向终点的向量, 这样 \mathbf{F} 的方向也是从起点指向终点的方向. 这表明, 我们规定, 杆上终点处的力是正向的. 给定网格中的点 \mathbf{p} , 连接它的杆的端点包含起点和终点两类, 起点杆的力为负, 终点杆的力为正. 由此可知, 合力为

```
1 Ftot = full(sparse(bars(:,[1,1,2,2]),ones(size(L))*[1,2,1,2],[-F,F],N,2));
```

这里, $\text{bars}(:, [1,1,2,2])$ 的 1,2 分别表示杆的起点和终点, $\text{ones}(\text{size}(L))*[1,2,1,2]$ 中的 1,2 分别表示向量的 x 和 y 分量. 注意, $\text{sparse}(i, j, s, m, n)$ 中的 i, j, s 可以是相同维数的矩阵, 相当于全部拉直.

固定点的合力设为零, 因为我们不想它移动, 于是新的散点为

```
1 Ftot(1:size(pfix,1),:) = 0; % Force = 0 at fixed points
2 p = p + dt*Ftot;
```

边界点的处理

节点更新的过程中, 部分点可能超出边界, 为此将它们拉回为最近的边界点. 由 (3.1) 知, 最近的边界点为 $\mathbf{p} - d_\Omega(\mathbf{p})\nabla d_\Omega(\mathbf{p})$.

```
1 % 5. Bring outside points back to the boundary
2 d = fd(p); ix = d>0; % Find points outside (d>0)
3 n1 = fd([p(ix,1)+deps,p(ix,2)]-d(ix))/deps;
4 n2 = fd([p(ix,1),p(ix,2)+deps]-d(ix))/deps;
5 p(ix,:) = p(ix,:)-[d(ix).*n1,d(ix).*n2]; % Project back to the boundary
```

对以上过程循环即可获得质量很好的网格. 迭代时给定最大迭代次数, 在此基础上, 当 $\Delta t \mathbf{F}(\mathbf{p}_n)$ 变化不大时, 也停止迭代.

注 3.4 差分计算得到的法向量不一定满足模 1 (对椭圆例子发现的确如此), 为此修改如下

```
1 % 5. Bring outside points back to the boundary
2 d = fd(p); ix = d>0; % Find points outside (d>0)
3 n1 = fd([p(ix,1)+deps,p(ix,2)]-d(ix))/deps;
4 n2 = fd([p(ix,1),p(ix,2)+deps]-d(ix))/deps;
5 nn = sqrt(n1.^2 + n2.^2);
6 n1 = n1./nn; n2 = n2./nn;
7 p(ix,:) = p(ix,:)-[d(ix).*n1,d(ix).*n2]; % Project back to the boundary
```

多次重复上面的过程可改善反射投影 (特别是高维问题).

3.2.4 简化版程序

```
1 function [p,t] = distmesh(fd,h0,BdBox,pfix)
2
3 if nargin==3, pfix = []; end
4
5 geps = 0.001*h0; Fscale = 1.2; dt = 0.2;
6 deps = sqrt(eps)*h0; dptol = 1e-3;
7
8 % 1. Create initial distribution in bounding box (equilateral triangles)
9 x = BdBox(1):h0:BdBox(2); y = BdBox(3):h0*sqrt(3)/2:BdBox(4);
10 [x,y] = meshgrid(x,y);
11 x(2:2:end,:) = x(2:2:end,:)+h0/2; % Shift even rows
12 p = [x(:,1),y(:,1)]; % List of node coordinates
13
14 % 2. Remove points outside the region
15 p = p(fd(p)<geps,:); % Keep only d<0 points
16 if ~isempty(pfix), p = setdiff(p,pfix,'rows'); end % Remove duplicated nodes
17 p = [pfix; p];
18 N = size(p,1);
19
20 iter = 0; MaxIter = 1e3;
21 while iter < MaxIter
22     iter = iter+1;
23     % 3. Get the truss and bars by the Delaunay algorithm
24     t = delaunay(p);
25     pm = (p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3; % Compute centroids
26     t = t(fd(pm)<-geps,:); % Keep interior triangles
27     totalEdge = sort([t(:,[2 3]); t(:,[3 1]); t(:,[1 2])],2);
28     bars = unique(totalEdge,'rows'); % Bars as node pairs
29
30     % 4. Update the points
31     barvec = p(bars(:,2),:)-p(bars(:,1),:); % List of bar vectors
32     L = sqrt(sum(barvec.^2,2)); % Bar lengths
33     L0 = Fscale*sqrt(sum(L.^2)/length(L)); % Desired lengths
34     f = max(L0-L,0); % Bar forces (scalars)
35     F = f./L*[1,1].*barvec; % Bar forces (x,y components)
36     % Force resultant
37     Ftot = full(sparse(bars(:,[1,1,2,2]),ones(size(L))*[1,2,1,2],[-F,F],N,2));
38     Ftot(1:size(pfix,1),:) = 0; % Force = 0 at fixed points
39     p = p + dt*Ftot;
40
41     % 5. Bring outside points back to the boundary
42     d = fd(p); ix = d>0; % Find points outside (d>0)
43     n1 = (fd([p(ix,1)+deps,p(ix,2)])-d(ix))/deps;
44     n2 = (fd([p(ix,1),p(ix,2)+deps])-d(ix))/deps;
45     nn = sqrt(n1.^2 + n2.^2);
46     n1 = n1./nn; n2 = n2./nn;
47     p(ix,:)=p(ix,:)-[d(ix).*n1,d(ix).*n2]; % Project back to the boundary
```

```

48
49      % 6. Termination criterion: All interior nodes move less than dptol (scaled)
50      if max(sqrt(sum(dt*Ftot(d<-geps,:).^2,2))/h0)<dptol, break; end
51  end
52
53 % final mesh
54 t = delaunay(p);
55 pc = (p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3;
56 t = t(fd(pc)<-geps,:);

```

注 3.5 要注意的是, 循环生成的 p 要再进行一次 Delaunay 三角剖分, 以实现 p 和 t 的对应.

3.2.5 添加 mesh size function

Mesh size function $h(x, y)$ 在网格自适应中经常使用, 它用来指定几何中不同位置的网格边长的尺度分布. 例如取 $h(x, y) = 1 + x$, 期望的结果是靠近 $x = 0$ 的尺度为 1, 而靠近 $x = 1$ 的尺度为 2, 这里的尺度不是真实尺度, 只是相对分布.

下面将改动前面的程序以实现这种目的, 显然只需要修改

$$f(\ell, \ell_0) = \begin{cases} k(\ell_0 - \ell), & \ell < \ell_0, \\ 0, & \ell \geq \ell_0 \end{cases}$$

中的期望长度 ℓ_0 . 在简化程序中, 我们取 ℓ_0 为迭代过程中所有杆的平均长度乘以 1.2, 即

$$\ell_0 = 1.2 \times \sqrt{\frac{L_1^2 + \dots + L_n^2}{n}}.$$

根据 mesh size function 的含义, 要把期望长度乘以 $h(x, y)$ 对应的尺度分布, 即

$$\ell_0 \rightarrow \bar{\ell}_0 = \frac{h}{h_{mean}} \ell_0 = 1.2 \times \frac{h}{\sqrt{\frac{h_1^2 + \dots + h_n^2}{n}}} \times \sqrt{\frac{L_1^2 + \dots + L_n^2}{n}} = 1.2 \times h \sqrt{\frac{L_1^2 + \dots + L_n^2}{h_1^2 + \dots + h_n^2}}.$$

为了方便, h 取为杆中点处的值.

另外, 对尺度大的地方, 散点不必很多, 为此 DistMesh 中增加了初始散点的拒绝处理. 对符号距离函数和 mesh size function, 它们可能还依赖于固定点等, 如 $d = d(p, p_{fix})$ (如多边形区域). 为此, 程序中使用 feval 函数 $d = \text{feval}(fd, p, \text{varargin} :)$ 来处理可变自变量个数的情形. 修改后的程序如下.

```

1 function [p,t] = distmesh2d(fd,fh,h0,BdBox,pfix,varargin)
2
3 if nargin==4, pfix = []; end
4

```

```

5 geps = 0.001*h0; Fscale = 1.2; dt = 0.2;
6 deps = sqrt(eps)*h0; dptol = 1e-3;
7
8 % 1. Create initial distribution in bounding box (equilateral triangles)
9 x = BdBox(1):h0:BdBox(2); y = BdBox(3):h0*sqrt(3)/2:BdBox(4);
10 [x,y] = meshgrid(x,y);
11 x(2:2:end,:) = x(2:2:end,:)+h0/2; % Shift even rows
12 p = [x(:,y(:)); % List of node coordinates
13
14 % 2. Remove points outside the region, apply the rejection method
15 p = p(feval(fd,p,varargin{:})<geps,:); % Keep only d<0 points
16 r0 = 1./feval(fh,p,varargin{:}).^2; % Probability to keep point
17 p = p(rand(size(p,1),1)<r0./max(r0),:); % Rejection method
18 if isempty(pfix), p = setdiff(p,pfix,'rows'); end % Remove duplicated nodes
19 pfix = unique(pfix,'rows');
20 p = [pfix; p];
21 N = size(p,1);
22
23 iter = 0; MaxIter = 1e3;
24 while iter < MaxIter
25     iter = iter+1;
26     % 3. Get the truss and bars by the Delaunay algorithm
27     t = delaunay(p);
28     pc = (p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3; % Compute centroids
29     t = t(feval(fd,pc,varargin{:})<-geps,:); % Keep interior triangles
30     totalEdge = sort([t(:,[2,3]); t(:,[3,1]); t(:,[1,2])],2);
31     bars = unique(totalEdge,'rows'); % Bars as node pairs
32
33     % 4. Update the points
34     barvec = p(bars(:,2),:)-p(bars(:,1),:); % List of bar vectors
35     L = sqrt(sum(barvec.^2,2)); % L = Bar lengths
36     pm = (p(bars(:,1),:)+p(bars(:,2),:))/2;
37     hbars = feval(fh,pm,varargin{:});
38     L0 = Fscale*hbars*sqrt(sum(L.^2)/sum(hbars.^2)); % Desired lengths
39     f = max(L0-L,0); % Bar forces (scalars)
40     F = f./L*[1,1].*barvec; % Bar forces (x,y components)
41     % Force resultant
42     Ftot = full(sparse(bars(:,[1,1,2,2]),ones(size(L))*[1,2,1,2],[-F,F],N,2));
43     Ftot(1:size(pfix,1),:) = 0; % Force = 0 at fixed points
44     p = p + dt*Ftot;
45
46     % 5. Bring outside points back to the boundary
47     d = feval(fd,p,varargin{:}); ix=d>0; % Find points outside (d>0)
48     n1 = (feval(fd,[p(ix,1)+deps,p(ix,2)],varargin{:})-d(ix))/deps;
49     n2 = (feval(fd,[p(ix,1),p(ix,2)+deps],varargin{:})-d(ix))/deps;
50     p(ix,:) = p(ix,:)-[d(ix).*n1,d(ix).*n2]; % Project back to the boundary
51
52     % 6. Termination criterion: All interior nodes move less than dptol (scaled)
53     if max(sqrt(sum(dt*Ftot(d<-geps,:).^2,2))/h0)<dptol, break; end
54 end

```

```
55
56 % final mesh
57 t = delaunay(p);
58 pc = (p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3;
59 t = t(feval(fd,pc,varargin{:})<-geps,:);
```

注 3.6 如果忽略 Delaunay 三角剖分, 那么在迭代过程中, 上一次的节点和移动后的节点是对应的. 通常在网格节点移动幅度不是很大时, Delaunay 三角剖分不必进行. 反之, 若仍采用移动前后的对应, 则三角剖分的拓扑结构会很差. 为了节省计算时间, 可增加另一个循环判断, 仅对变化幅度大的情形再进行 Delaunay 三角剖分, 这里省略程序.

一些例子见 distmesh_benchmark2d.m.

3.3 DistMesh3D

DistMesh 的网格优化思想适合任意维问题, 难点还是在于建立 Delaunay 三角剖分. 对大于或等于三维的情形, 本文暂不考虑 Delaunay 三角剖分算法.

Part II

经典问题的有限元方法

第四章 一维问题的有限元方法

以如下的混合边值问题作为本章的模型问题

$$\begin{cases} -u'' + cu = f(x), & 0 < x < 1, \\ u(0) = 0, \quad u'(1) = 0, \end{cases} \quad (4.1)$$

其中 $c > 0$ 是常数. 变分问题为: 找 $u \in V$, 使得

$$a(u, v) = \ell(v) \quad \forall v \in V, \quad (4.2)$$

式中,

$$\begin{aligned} a(u, v) &= \int_0^1 (u'v' + cuv)dx, \quad \ell(v) = (f, v) = \int_0^1 fvdx, \\ V &= \{v \in L^2(0, 1) : a(v, v) < \infty, \quad v(0) = 0\}. \end{aligned}$$

4.1 单元与整体的关系

有限元编程中最重要的就是如何从单元刚度矩阵和单元载荷向量获得整体刚度矩阵和整体载荷向量, 我们称这个过程为装配. 为了实现装配, 首先必须弄清楚单元刚度矩阵或载荷向量与整体刚度矩阵或载荷向量的关系.

有限元方法是基于变分形式的数值方法, 在乘以检验函数并分部积分时, 就不可避免地要遇到边界项. 例如, 对模型问题 (4.1), 在不考虑边界条件的情况下, 变分形式应该为

$$\int_0^1 (u'v' + cuv)dx - u'|_0^1 = \int_0^1 f(x)v(x)dx.$$

对高维问题, $-u'|_0^1$ 对应边界积分. 要注意, 对不同的边界条件, 边界积分可能含有未知量, 对模型问题 (4.1), 它恰好为零. 为了方便, 我们统称边界积分项和边界条件为边界项, 而且为了一般性, 有限元编程的一个重要原则是: 最后处理边界项. 为了突出省略了边界项, 变分形式写为

$$\int_0^1 (u'v' + cuv)dx \sim \int_0^1 f(x)v(x)dx,$$

符号 “ \sim ” 表示省略了边界项.

4.1.1 整体节点基与局部节点基

设所考虑问题的区域为 $\Omega = (0, 1)$, 给定划分 $0 = x_0 < x_1 < \dots < x_n = 1$, 设 $K_j = [x_{j-1}, x_j]$, $j = 1, 2, \dots, n$, 则可引入分段线性函数空间 (不包含边界条件)

$$V_h = \{v \in C(\bar{\Omega}) : v \text{ 在每个 } K_j = [x_{j-1}, x_j] \text{ 上连续且为一次函数}, j = 1, 2, \dots, n\}.$$

我们知道, 每个节点 x_i 都对应一个节点基 $\Phi_i(x)$, 满足 $\Phi_i(x_j) = \delta_{ij}$, $0 \leq i, j \leq n$, 它们合起来构成空间 V_h 的一组基. 这些基的图像如下图所示

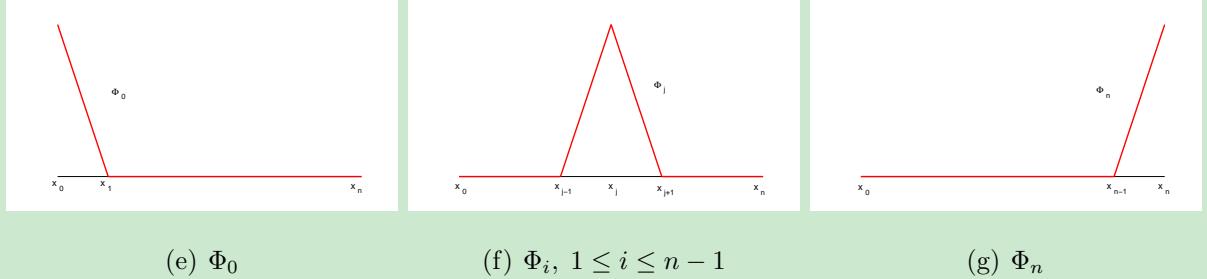


图 4.1. 整体基函数 $\Phi_i, i = 0, 1, \dots, n$

如果把整个区域 $\bar{\Omega}$ 换成单元 $K = [x_{j-1}, x_j]$, 那么可得两个节点基 ϕ_1, ϕ_2 , 其图像如下图所示

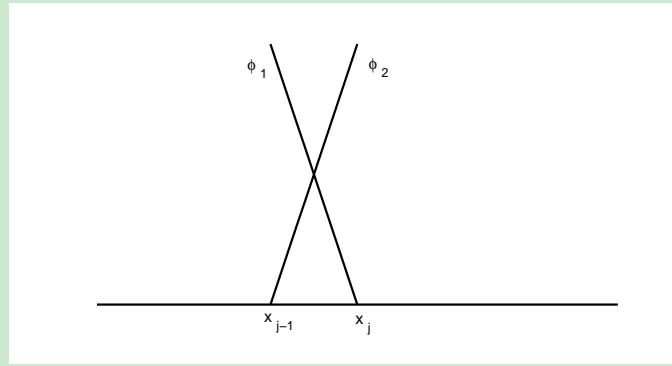


图 4.2. 单元 K 的节点基

我们称 ϕ_1, ϕ_2 为局部节点基.

一个重要的关系就是, 整体节点基限制在单元上就是局部节点基. 例如,

$$\Phi_{j-1}|_K = \phi_1, \quad \Phi_j|_K = \phi_2.$$

之所以成立, 是因为 $\Phi_{j-1}|_K$ 是一次函数, 且 $\Phi_{j-1}|_K(x_{j-1}) = 1$, $\Phi_{j-1}|_K(x_j) = 0$. 这个关系对高维问题仍成立, 正是有这一条性质才导致有限元可以很方便地按单元考虑.

4.1.2 整体刚度矩阵与单元刚度矩阵的关系

在不考虑边界项时, 整体刚度矩阵可由单元刚度矩阵获得, 我们以模型问题 (4.1) 来说明装配的过程.

步 1: 第 j 个方程及其单元分解

设整体基函数为 $\Phi_0, \Phi_1, \dots, \Phi_n$ (视 u_0 也为变量), 待求函数表为

$$u = \sum_{i=0}^n u_i \Phi_i,$$

则系统方程组的第 j ($= 0, 1, \dots, n$) 个方程就是在 (4.2) 中取 $v = \Phi_j$. 按单元求和, 左端和右端分别为

$$\begin{aligned} a(u, v) &= \int_0^1 (u'v' + cuv) dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + cuv) dx, \quad v = \Phi_j, \\ \ell(v) &= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} fv dx, \quad v = \Phi_j. \end{aligned}$$

为此, 可以先考虑单元上的部分, 即

$$\begin{aligned} a(u, v)_{K_i} &= \int_{x_{i-1}}^{x_i} (u'v' + cuv) dx, \quad v = \Phi_j, \\ \ell(v)_{K_i} &= \int_{x_{i-1}}^{x_i} fv dx, \quad v = \Phi_j, \end{aligned}$$

它们是第 j 个方程的组成成分, 加起来的过程就是合并同类项.

步 2: 单元上的形式

设单元 K_i 上两个局部节点基为 ϕ_1 和 ϕ_2 , 则 u 限制在 K_i 就是 $u_{i-1}\phi_1 + u_i\phi_2$, 即

$$u = u_{i-1}\phi_1 + u_i\phi_2, \quad x \in K_i,$$

于是

$$a(u, v)_{K_i} = \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)'v' + c(u_{i-1}\phi_1 + u_i\phi_2)v] dx, \quad v = \Phi_j.$$

在逐个代入 $v = \Phi_j$ 时, 单元 K_i 上有贡献的只有 Φ_{i-1} 和 Φ_i , 局部上恰好对应 ϕ_1 和 ϕ_2 , 也就是说单元 K_i 上的积分实际上只在第 $i-1$ 个方程和第 i 个方程上有贡献, 它们分别为

$$\begin{aligned} v = \Phi_{i-1} : \quad \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)' \phi'_1 + c(u_{i-1}\phi_1 + u_i\phi_2)\phi_1] dx &\sim \int_{x_{i-1}}^{x_i} f\phi_1 dx, \\ v = \Phi_i : \quad \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)' \phi'_2 + c(u_{i-1}\phi_1 + u_i\phi_2)\phi_2] dx &\sim \int_{x_{i-1}}^{x_i} f\phi_2 dx, \end{aligned}$$

写成矩阵形式为

$$\int_{x_{i-1}}^{x_i} \begin{bmatrix} \phi'_1 \phi'_1 + c\phi_1 \phi_1 & \phi'_2 \phi'_1 + c\phi_2 \phi_1 \\ \phi'_1 \phi'_2 + c\phi_1 \phi_2 & \phi'_2 \phi'_2 + c\phi_2 \phi_2 \end{bmatrix} dx \begin{bmatrix} u_{i-1} \\ u_i \end{bmatrix} \sim \int_{x_{i-1}}^{x_i} \begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix} dx.$$

可以看到左边的矩阵恰是单元刚度矩阵, 右边的向量恰是单元载荷向量. 注意矩阵的第一行对应 $v = \Phi_{i-1}$, 它应加到第 $i-1$ 个系统方程中, 第二行对应 $v = \Phi_i$, 它应加到第 i 个系统方程中, 与这里的变量 $[u_{i-1}, u_i]^T$ 正好对应.

步 3: 遍历所有单元

当遍历所有单元后, 就得到系统方程组, 而遍历的过程恰好是把单元刚度矩阵的分量加到整体刚度矩阵的对应位置.

4.2 刚度矩阵与载荷向量的装配

4.2.1 载荷向量的装配

我们已经看到, 装配的过程就是把单元刚度矩阵或载荷向量的元素加到整体刚度矩阵或载荷向量的正确位置, 这就需要局部与整体编号的一种对应.

为了方便, 我们先考虑载荷向量的装配. 设单元 $K_i = [x_{i-1}, x_i]$ ($i = 1, 2, \dots, n$) 上的两个局部节点基为 ϕ_1, ϕ_2 , 单元载荷向量为

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix}.$$

对单元载荷向量, 可如下分析:

a) ϕ_1 在 K_i 上对应的恰是左端点的整体节点基 Φ_{i-1} , ϕ_2 在 K_i 上对应的恰是右端点的整体节点基 Φ_i , 于是单元载荷向量用整体基函数表示为

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix} = \begin{bmatrix} \ell_{K_i}(\Phi_{i-1}) \\ \ell_{K_i}(\Phi_i) \end{bmatrix}.$$

b) $\ell_{K_i}(\Phi_{i-1})$ 是 $\ell(\Phi_{i-1})$ 的一部分, 自然贡献给 $F_{i-1} = \ell(\Phi_{i-1})$;

$\ell_{K_i}(\Phi_i)$ 是 $\ell(\Phi_i)$ 的一部分, 自然贡献给 $F_i = \ell(\Phi_i)$.

c) 因此有如下贡献关系

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix} \rightarrow \begin{bmatrix} F_{i-1} \\ F_i \end{bmatrix}.$$

综上, 我们可以获得如下的装配算法.

1. 初始化载荷向量 F 为零向量, 注意行对应 u_0, u_1, \dots, u_n , 设单元编号 $i = 1$.
2. 计算相应的单元载荷 F_{K_i} .
3. 把单元向量的分量加入到整体载荷向量的对应位置

$$\begin{aligned} F_{i-1} &\leftarrow F_{i-1} + F_{K_i}(1), \\ F_i &\leftarrow F_i + F_{K_i}(2). \end{aligned}$$

4. $i \leftarrow i + 1$, 转到第 2 步, 直到 $i = n + 1$ 结束.
-

上面的装配过程表明有如下的局部与整体对应

index : $\{1, 2\}$ (local) \rightarrow $\{i-1, i\}$ (global), 对单元 K_i ,

即

$$\text{index}(1) = i - 1, \quad \text{index}(2) = i,$$

我们称 index 为装配指标. 显然, 局部整体对应与单元的连通性是一致的.

4.2.2 刚度矩阵的装配

当 x_i, x_j 不相邻时, $K_{ij} = 0$, 故 K 是一个稀疏矩阵(且对称). 设单元 $K_i = [x_{i-1}, x_i]$ ($i = 1, 2, \dots, n$) 的两个节点基为 ϕ_1, ϕ_2 , 对应的单元刚度矩阵为

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix}.$$

对单元刚度矩阵, 如下分析:

a) ϕ_1 在 K_i 上对应的恰是 Φ_{i-1} , ϕ_2 在 K_i 上对应的恰是 Φ_i , 于是单元刚度矩阵用整体基函数表示为

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix} = \begin{bmatrix} a(\Phi_{i-1}, \Phi_{i-1})_{K_i} & a(\Phi_{i-1}, \Phi_i)_{K_i} \\ a(\Phi_i, \Phi_{i-1})_{K_i} & a(\Phi_i, \Phi_i)_{K_i} \end{bmatrix}.$$

b) $a(\Phi_{i-1}, \Phi_{i-1})_{K_i}$ 是 $a(\Phi_{i-1}, \Phi_{i-1})$ 的一部分, 自然贡献给 $K_{i-1,i-1} = a(\Phi_{i-1}, \Phi_{i-1})$;

$a(\Phi_{i-1}, \Phi_i)_{K_i}$ 是 $a(\Phi_{i-1}, \Phi_i)$ 的一部分, 自然贡献给 $K_{i-1,i} = a(\Phi_{i-1}, \Phi_i)$;

$a(\Phi_i, \Phi_{i-1})_{K_i}$ 是 $a(\Phi_i, \Phi_{i-1})$ 的一部分, 自然贡献给 $K_{i,i-1} = a(\Phi_i, \Phi_{i-1})$;

$a(\Phi_i, \Phi_i)_{K_i}$ 是 $a(\Phi_i, \Phi_i)$ 的一部分, 自然贡献给 $K_{i,i} = a(\Phi_i, \Phi_i)$.

c) 因此有如下贡献关系

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix} \rightarrow \begin{bmatrix} K_{i-1,i-1} & K_{i-1,i} \\ K_{i,i-1} & K_{i,i} \end{bmatrix}.$$

我们有如下装配算法.

刚度矩阵的装配

1. 初始化刚度矩阵 K 为零矩阵, 注意行对应 u_0, u_1, \dots, u_n , 设单元编号 $i = 1$.
2. 计算相应的单元刚度矩阵 A_{K_i} .
3. 把单元矩阵的分量加入到整体刚度矩阵的对应位置

$$\begin{aligned} K_{i-1,i-1} &\leftarrow K_{i-1,i-1} + A_{K_i}(1, 1), \\ K_{i-1,i} &\leftarrow K_{i-1,i} + A_{K_i}(1, 2), \\ K_{i,i-1} &\leftarrow K_{i,i-1} + A_{K_i}(2, 1), \\ K_{i,i} &\leftarrow K_{i,i} + A_{K_i}(2, 2). \end{aligned}$$

4. $i \leftarrow i + 1$, 转到第 2 步, 直到 $i = n + 1$ 结束.
-

注意到

$$\begin{bmatrix} K_{i-1,i-1} & K_{i-1,i} \\ K_{i,i-1} & K_{i,i} \end{bmatrix}$$

恰好在矩阵三对角位置上, 因此整体刚度矩阵是三对角的. 一般而言, 单元刚度矩阵不是对称的, 若是对称的, 则只要计算上三角部分.

4.2.3 指标装配法

鉴于 MATLAB 向量标号是从 1 开始, 以下也遵循这个规定, 并给出如下的 5 个线性单元剖分.

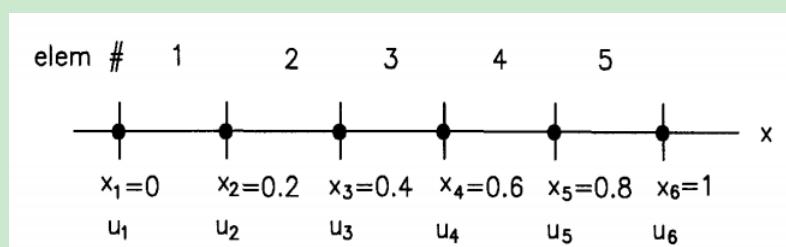


图 4.3. 5 个线性单元的剖分

设单元方程如下

$$\begin{bmatrix} k_{11}^i & k_{12}^i \\ k_{21}^i & k_{22}^i \end{bmatrix} \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} \sim \begin{bmatrix} F_i \\ F_{i+1} \end{bmatrix},$$

局部上, u_i 和 u_{i+1} 用 1 和 2 对应, 整体上则用 i 和 $i + 1$ 表示, 即

$$\{1, 2\} \text{ (local)} \quad \rightarrow \quad \{i, i + 1\} \text{ (global)},$$

于是可建立如下数组

$$\text{index}(1) = i, \quad \text{index}(2) = i + 1, \quad \text{对第 } i \text{ 个单元},$$

它实现了局部与整体的对应.

装配分两步进行, 即装配单元矩阵和单元向量. 对每个单元, 用 **ke** 表示单元矩阵, **fe** 表示单元向量. 用 **kk** 表示系统矩阵, **ff** 表示系统向量. 设单元矩阵和单元向量如下

$$\mathbf{ke} = \begin{bmatrix} \frac{1}{h_i} + \frac{h_i}{3} & -\frac{1}{h_i} + \frac{h_i}{6} \\ -\frac{1}{h_i} + \frac{h_i}{6} & \frac{1}{h_i} + \frac{h_i}{3} \end{bmatrix}, \quad \mathbf{fe} = \begin{bmatrix} \frac{h_i}{6}(2x_i + x_{i+1}) \\ \frac{h_i}{6}(2x_{i+1} + x_i) \end{bmatrix} \longleftrightarrow \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix}.$$

我们先装配单元向量. **fe** 的第一行对应系统的 i , 即 $\text{index}(1) = i$, **fe** 的第二行对应系统的 $i + 1$, 即 $\text{index}(2) = i + 1$. **f** 的第一行应加到 **ff** 的第 i 行, 第二行应加到 **ff** 的第 $i + 1$ 行. 为此可如下编写程序:

```

1 for iel = 1:nel          % 单元循环
2     index = elem(iel,:);% local-->global
3     for i = 1:2           % f 的行循环 (local)
4         ii = index(i);   % f 的行在系统中的编号 (global)
5         ff(ii) = ff(ii)+fe(i); % 把 f 的行放到系统向量的行中
6     end
7 end

```

ke 既涉及到行又涉及到列, 在前面装配 **fe** 的时候实际上已经确定了行, 现在需要确定列. 注意与 u_j 相乘的元素位于第 j 列, 即与指标 **index** 对应.

为此, 装备 **ke** 和 **fe** 的过程可如下编写程序:

```

1 for iel = 1:nel          % 单元循环
2     index = elem(iel,:);  % local-->global
3     for i = 1:2           % f 的行循环 (local)
4         ii = index(i);   % f 的行在系统中的编号 (global)
5         ff(ii) = ff(ii)+fe(i); % 把 f 的行放到系统向量的行中
6         for j = 1:2         % k 的列循环 (local)
7             jj = index(j); % k 的列在系统中的编号 (global)
8             kk(ii,jj) = kk(ii,jj)+ke(i,j);
9             % 把 k 的行列元素放到系统向量的行列中
10        end
11    end
12 end

```

MATLAB 支持矩阵用向量取多行或多列, 因而相加的过程可简单写为

```
1 ff(index) = ff(index)+fe;
2 kk(index,index) = kk(index,index)+ke;
```

这里 $kk(index, index)$ 包含交叉位置的元素. 本文称这种装配策略为指标法, 它的好处在于明确了局部与整体的对应, 便于直接推广. 事实上, 上面的装配程序可以直接平移到高阶有限元以及高维问题中, 只不过要相应修改 $index$ 罢了, 后面将会看到这一点.

上面的装配过程适合逐个单元进行, 即一边计算单元刚度矩阵和载荷向量, 一边进行装配. 一种更高效的装配策略是, 首先计算出所有单元刚度矩阵, 然后用 `sparse` 函数进行一次性装配. 对大规模问题, 后者更好. 尽管前者在一开始可声明为 `sparse` 矩阵, 但一般不提倡对稀疏矩阵进行运算.

4.2.4 `sparse` 装配法

刚度矩阵的装配

前面给出的装配为

```
1 kk(index, index) = kk(index, index)+ke;
```

它易于理解和推广, 但未充分利用 MATLAB 的向量运算. 事实上, 我们可以把所有的指标对应放在一起实现一次性装配, 本文称为 `sparse` 装配法. 注意, 它只是前面指标装配的再加工.

考虑 (4.4) 给出的单元刚度矩阵. 为了实现一次性装配, 我们要用 k_{ij} 存储所有单元 (i, j) 位置的结果, 这里的 (i, j) 是局部编号, 即

```
1 % ----- All element matrices -----
2 h = diff(node);
3 k11 = -acoef./h+bcoef/2*(-1)+ccoeff*h./6*2;
4 k12 = -acoef./h*(-1)+bcoef/2+ccoeff*h./6;
5 k21 = -acoef./h*(-1)+bcoef/2*(-1)+ccoeff*h./6;
6 k22 = -acoef./h+bcoef/2+ccoeff*h./6*2;
```

根据对称性, 上面只需存储上三角部分, 为了一般性, 这里全部存储.

设单元的左端点整体编号为 z_1 , 右端点为 z_2 , 我们要给出所有单元的 local-global 对应, 显然为

```
1 % ----- local --> global -----
2 z1 = elem(:,1); z2 = elem(:,2);
```

且前面给出的第 i_{el} 个单元的装配指标就是

```
index = [z1(iel), z2(iel)].
```

我们可按照下面的方式逐一装配.

```
1 % ----- Assemble the matrix -----
2 % upper triangular
3 kk = sparse(z1,z2,k12,N,N);
4 % lower triangular
5 kk = kk+sparse(z2,z1,k21,N,N);
6 % diagonal
7 kk = kk+sparse(z1,z1,k11,N,N);
8 kk = kk+sparse(z2,z2,k22,N,N);
```

如果是对称矩阵, 那么可如下编写

```
1 kk = sparse(z1,z2,k12,N,N);
2 kk = kk+kk';
3 kk = kk+sparse(z1,z1,k11,N,N);
4 kk = kk+sparse(z2,z2,k22,N,N);
```

这个装配方法比之前的指标法更快一点, 但仍有缺点:

1. 对高维问题, 单元刚度矩阵的分量较多, 逐一相加比较麻烦;
2. sparse 函数的快速在于建立稀疏矩阵, 做加减等运算时, 它还是要按稠密矩阵进行计算.

在优化之前, 我们先来分析一下上面的装配.

- 对固定单元, 设单元刚度矩阵为

$$A_K = \begin{array}{cc|c} u_i & u_{i+1} & \\ \hline k_{11} & k_{12} & u_i \\ k_{21} & k_{22} & u_{i+1} \end{array},$$

这里右侧表示整体刚度矩阵的行指标, 上侧表示列指标. 对 k_{12} , 我们要把它放在 $(i, i + 1)$ 位置, 按照前面编程的思路, 是 (z_1, z_2) 位置. 也就是说, 我们要在 (z_1, z_2) 处赋值 k_{12} , 这可用稀疏矩阵函数 sparse 完成, 其用法如下

```
S = sparse(i, j, s, m, n);
```

它分配了一个 $m \times n$ 的稀疏零矩阵, 其中 i, j 是向量, 对应分量指出非零值的位置, s 则是非零值的向量. 因此, 若想在 (z_1, z_2) 处赋值 k_{12} , 可以如下操作

```
A = sparse(z1, z2, k12, N, N);
```

注意, 该命令把所有单元刚度矩阵的 $(1,2)$ 处的值放到了整体刚度矩阵的对应位置中.

- 我们自然会有这样的疑问: 如果有两个单元刚度矩阵 (1,2) 处的值对应相同的整体指标 (i, j) , 那么它们应该相加, 而不是简单的赋值.

事实上, 在 MATLAB 中, `sparse` 函数有一个特殊性质 (summation property): 当出现相同指标 (i, j) 时, 规定非零值相加. 这样, 若的确出现上面的情况, 则本身已经解决.

我们要说的是, 对非对角线元素, 上面提到的情形是不可能出现的. 因为 K_i 对应 $\{u_i, u_{i+1}\}$, 只可能出现 $(i, i + 1), (i + 1, i)$, 而对 K_{i+1} 则为 $(i + 1, i + 2), (i + 2, i + 1)$, 两者没有共同位置.

- 上面最后处理对角线元素是防止对称情形转置相加时重复相加对角线元素.

`sparse` 函数的 summation property 使得我们可以进一步优化上面的装配过程, 即把所有位置的 (i, j, s) 拼接在一起实现, 即

$$\begin{bmatrix} i_{11} & j_{11} & s_{11} \\ i_{12} & j_{12} & s_{12} \\ i_{21} & j_{21} & s_{21} \\ i_{22} & j_{22} & s_{22} \end{bmatrix},$$

称其为 sparse 装配指标.

1. 我们按单元刚度矩阵行优先的顺序排列每列的指标;
2. i_{ij} 表示所有单元矩阵 (i, j) 位置的整体编号向量, 共有 nel 个元素, 其中 nel 是单元的个数;
3. 记上面的矩阵为 $[i, j, s]$, 则 i 共有 $nel \times N_{dof}^2$ 个元素, 其中, $N_{dof} = 2$ 表示单元的自由度个数.

核心代码如下

```

1 % ----- local --> global -----
2 Ndof = 2; nnz = nel*Ndof^2;
3 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = zeros(nnz,1);
4 id = 0;
5 for i = 1:Ndof
6     for j = 1:Ndof
7         ii(id+1:id+nel) = elem(:,i);    % zi
8         jj(id+1:id+nel) = elem(:,j);    % zj
9         ss(id+1:id+nel) = K(:,i,j);    % kij
10        id = id + nel;
11    end
12 end

```

这里假设 (i, j) 位置的单元刚度矩阵值用三维数组存储.

- `elem` 按行存储单元的好处是可以按列操作同一个顶点.
- 可用向量法给出 `ii`, `jj`, 只需要两行语句, 即

```
1 ii = reshape(repmat(elem, Ndof, 1), [], 1);
2 jj = repmat(elem(:, ), Ndof, 1);
```

以后都采用这种方式.

- 我们将采用另一种方式存储单元矩阵, 即按行拉直存储所有单元的结果

$$K = [k_{11}, k_{12}, k_{21}, k_{22}].$$

这里, k_{11} 是所有单元给出的向量, 其他位置类似, 且不妨称 K 为行拉直矩阵. 显然 `ss` 由该矩阵按列拉直得到. **后面将采用这种处理, 为此在生成单元刚度矩阵时, 我们实施行拉直.**

可以如下给出装配算法

CODE 4.1. sparse 装配

```
1 K = [k11, k12, k21, k22]; % stored in rows
2 Ndof = 2;
3 ii = reshape(repmat(elem, Ndof, 1), [], 1);
4 jj = repmat(elem(:, ), Ndof, 1);
5 ss = K(:, );
6 kk = sparse(ii, jj, ss, N, N);
```

载荷向量的装配

单元载荷向量为

$$[F_K] = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}, \quad F_i = \int_K f v dx dy, \quad v = \phi_i, \quad i = 1, 2,$$

用中心格式近似有

$$[F^e] = \int_K \begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix} dx dy \approx \left[\begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix} \right]_{x_c} \cdot |K| = f(x_c) \cdot \frac{h_i}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

显然有

$$F_1 = F_2 = f(x_c) \cdot \frac{h_i}{2}.$$

按前面逐一装配的方式可如下进行

```

1 % ----- Assemble the vector -----
2 x1 = node(1:N-1); x2 = node(2:N); xc = (x1+x2)./2;
3 fi = f(xc).*h./2;
4 F1 = fi; F2 = fi;
5 ff = sparse(z1,1,F1,N,1);
6 ff = ff+spars(z2,1,F2,N,1);

```

根据 `sparse` 的 summation property, 最后的两句实际上可直接写为

```
1 ff = sparse([z1;z2],1,[F1;F2],N,1);
```

注意到 $[z_1; z_2] = \text{elem}(:)$, 也可写为

```
1 ff = sparse(elem(:,1,[F1;F2],N,1);
```

`ff` 一般不是稀疏的, 存储为稀疏的形式访问起来会麻烦. 我们可用下面的语句代替

```
1 ff = accumarray(elem(:,1,[F1;F2],N,1));
```

同理, 右端向量也按行拉直存储, 即

```

1 % ----- Assemble the vector -----
2 x1 = node(1:N-1); x2 = node(2:N);
3 xc = (x1+x2)./2;
4 F1 = f(xc).*h./2; F2 = F1; F = [F1,F2];
5 ff = accumarray(elem(:,1,[F1;F2],N,1));

```

`accumarray` 是累加函数, 用法如下.

1. 若 `accumarray` 有两个参数, 则第一个参数是数组的位置索引, 第二个参数是累加的数据. 例如

```

1 subs = [1; 2; 4; 2; 4]; vals = 101:105;
2 A = accumarray(subs,vals);

```

`subs` 必须是列向量, 其中的最大数为 4 表明 `A` 是 4 个元素的数组且初始为零数组; 索引 1 对应的值为 101, 则 `A` 的第 1 个位置累加 101; 索引 2 对应的有两个, 分别为 102, 104, 它们累加起来为 206; 索引 3 没有则仍为 0; 索引 4 对应有 103, 105, 累加起来为 208. 于是 `A = [101; 206; 0; 208]`.

特别地, 当 `vals = 2` 为单个数时, 默认为 `vals = vals*ones(max(subs),1)`, 此时 `A = [2; 4; 0; 4]`.

2. 位置索引可以是矩阵. 例如

```

1 subs = [1 1;
2          2 1;

```

```

3      2 3;
4      2 1;
5      2 3];
6 vals = 101:105
7 A = accumarray(subs,vals);

```

`subs` 列的最大值分别为 2,3, 则 `A` 是 2×3 的矩阵; `subs` 的行与 `vals` 的行对应; `subs` 的第一行是 $(1, 1)$ 表示矩阵的该位置累加 101, 第二行是 $(2, 1)$ 表示矩阵的该位置累加 102. 根据这个规律, 我们有 $A = [101 \ 0 \ 0; \ 206 \ 0 \ 208]$.

再考虑三维数组的例子.

```

1 subs = [1 1 1;
2      2 1 2;
3      2 3 2;
4      2 1 2;
5      2 3 2];
6 vals = 101:105;
7 A = accumarray(subs,vals);

```

`subs` 列的最大值分别为 2,3,3, 则 `A` 是 $2 \times 3 \times 2$ 的矩阵; 第一行为 $(1, 1, 1)$, 则矩阵的 $(1, 1, 1)$ 累加 101, 类似其他行. 这样, 我们有

```

A(:,:,1) =
101      0      0
      0      0      0

A(:,:,2) =
      0      0      0
206      0     208

```

3. 上面是通过列的最大值来确定矩阵 `A` 的维数, 我们也可自行设定矩阵的维数. 例如

```

1 subs = [1 1;
2      2 1;
3      2 3;
4      2 1;
5      2 3];
6 vals = 1;
7 A = accumarray(subs, vals, [2 4]);

```

若没有第三个参数 `[2 4]` (必须是行向量), 则 `A` 是 2×3 的矩阵 $[1 \ 0 \ 0; \ 2 \ 0 \ 2]$. 加上 `[2 4]`, 则是 2×4 的矩阵, 分析一致, 此时结果为 $[1 \ 0 \ 0 \ 0; \ 2 \ 0 \ 2 \ 0]$.

根据上面的分析, `ff = accumarray(elem(:,), F(:,), [N 1])` 是生成一个 $N \times 1$ 的零向量 `ff`, 且在 `elem(i)` 的位置累加 `F(i)`.

4.3 程序设计

4.3.1 问题说明

例 4.1 考虑更一般的两点边值问题

$$-au'' + bu' + cu = f(x), \quad 0 < x < L,$$

取精确解为

$$u(x) = \frac{1}{2e} e^{2x} - \frac{1}{2}(1 + e^{-1})e^x + \frac{1}{2}.$$

边界条件可以是 Dirichlet 边界条件或 Neumann 边界条件.

变分形式为

$$a(u, v) = \ell(v),$$

其中

$$\begin{aligned} a(u, v) &= \int_0^L (au'v' + bu'v + cuv)dx, \\ \ell(v) &= \int_0^L f(x)v(x)dx + au'v|_0^L. \end{aligned} \tag{4.3}$$

单元刚度矩阵为

$$\begin{aligned} [K^e] &= \int_{x_i}^{x_{i+1}} \left(a \begin{bmatrix} \phi'_1 \\ \phi'_2 \end{bmatrix} [\phi'_1, \phi'_2] + b \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} [\phi'_1, \phi'_2] + c \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} [\phi_1, \phi_2] \right) dx \\ &= \frac{a}{h_i} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + \frac{b}{2} \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} + \frac{ch_i}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \end{aligned} \tag{4.4}$$

单元载荷向量为

$$[F^e] = \int_{x_i}^{x_{i+1}} f(x) \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} dx \approx \frac{f(x_c)h_i}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

其中 x_c 是单元 $K_i = [x_i, x_{i+1}]$ 的中点, 即 $x_c = (x_i + x_{i+1})/2$.

在后面编程中, 我们主要用到如下的数据信息.

1. 节点编号及坐标

我们将用 `node` 表示所有节点的坐标, `node` 的索引即为节点编号.

2. 连通性

连通性指的是单元的顶点编号, 我们用 elem1D 表示, 它的第一列表示所有单元的第一个顶点编号, 第二列是第二个顶点编号. 由连通性立刻可以获得局部与整体对应的装配指标 index.

PDE 的数据如下生成.

```
1 function pde = pde1D(para)
2
3 syms x;
4 c1 = 0.5/exp(1); c2 = -0.5*(1+1/exp(1));
5 u = c1*exp(2*x)+c2*exp(x)+1/2;
6 % exact solution
7 uexact = eval(['@(x)',vectorize(u)]); % transform to anonymous function
8
9 % rhs
10 f = -para.a*diff(u,2)+para.b*diff(u,1)+para.c*u;
11 f = eval(['@(x)',vectorize(f)]);
12
13 % boundary conditions
14 du = diff(u); du = eval(['@(x)',vectorize(du)]);
15 Du = @(x) du(x); g_D = @(x) uexact(x);
16
17 pde = struct('f', f, 'uexact', uexact, 'g_D', g_D, 'Du', Du, 'para', para);
```

4.3.2 刚度矩阵和载荷向量的装配

刚度矩阵如下计算和装配.

```
1 %% Assemble stiffness matrix
2 % All element matrices
3 para = pde para;
4 a = para.a; b = para.b; c = para.c;
5 x1 = node(elem(:,1)); x2 = node(elem(:,2));
6 h = x2-x1;
7 k11 = a./h+b/2*(-1)+c*h./6*2;
8 k12 = a./h*(-1)+b/2+c*h./6;
9 k21 = a./h*(-1)+b/2*(-1)+c*h./6;
10 k22 = a./h+b/2+c*h./6*2;
11 K = [k11,k12,k21,k22]; % stored in rows
12 % stiffness matrix
13 ii = reshape(repmat(elem, Ndof,1), [], 1);
14 jj = repmat(elem(:, Ndof, 1);
15 kk = sparse(ii,jj,K(:,N,N));
```

载荷向量为

```

1 %% Assemble load vector
2 xc = (x1+x2)./2;
3 F1 = pde.f(xc).*h./2; F2 = F1; F = [F1,F2];
4 ff = accumarray(elem(:,), F(:,), [N 1]);

```

4.3.3 边界项的处理

下面说明边界项为什么可以最后处理, 我们以一个具体的例子重述前面的过程.

考虑方程

$$-u'' + u = x, \quad 0 < x < 1,$$

这里先不管边界条件. 对应的变分形式为

$$a(u, v) = \ell(v),$$

其中

$$a(u, v) = \int_0^1 (u'v' + uv)dx, \quad \ell(v) = \int_0^1 xvdx + u'v|_0^1.$$

设区间划分为 $0 = x_0 < x_1 < \dots < x_n = 1$, 记单元 $K_j = [x_{j-1}, x_j]$, $j = 1, 2, \dots, n$, $h_j = x_j - x_{j-1}$. 其上的插值基函数为 (统一记下标为 1,2)

$$\phi_1(x) = \frac{x_j - x}{h_j}, \quad \phi_2(x) = \frac{x - x_{j-1}}{h_j}.$$

变分形式写成单元累加形式为

$$\sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + uv)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} xvdx + u'v|_0^1. \quad (4.5)$$

先考虑

$$I = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + uv)dx - \sum_{i=1}^n \int_{x_{i-1}}^{x_i} xvdx,$$

单元 K_i 对应的是

$$I_i = \int_{x_{i-1}}^{x_i} (u'v' + uv)dx - \int_{x_{i-1}}^{x_i} xvdx.$$

把插值近似 $u = u_{i-1}\phi_1 + u_i\phi_2$ 和 $v = \phi_1, \phi_2$ 分别代入上式, 经过简单计算有离散形式

$$\tilde{I}_i = \begin{bmatrix} \frac{1}{h_{i-1}} + \frac{h_{i-1}}{3} & -\frac{1}{h_{i-1}} + \frac{h_{i-1}}{6} \\ -\frac{1}{h_{i-1}} + \frac{h_{i-1}}{6} & \frac{1}{h_{i-1}} + \frac{h_{i-1}}{3} \end{bmatrix} \begin{bmatrix} u_{i-1} \\ u_i \end{bmatrix} - \begin{bmatrix} \frac{h_{i-1}}{6}(2x_{i-1} + x_i) \\ \frac{h_{i-1}}{6}(2x_i + x_{i-1}) \end{bmatrix}.$$

若现在只有三个等分单元, 即 $x_0 = 0, x_1 = 1/3, x_2 = 2/3$ 和 $x_3 = 1$, 则对单元 1, 有

$$\tilde{I}_1 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} - \begin{bmatrix} 0.0185 \\ 0.0370 \end{bmatrix},$$

对单元 2, 有

$$\tilde{I}_2 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - \begin{bmatrix} 0.0741 \\ 0.0926 \end{bmatrix},$$

对单元 3, 有

$$\tilde{I}_3 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0.1296 \\ 0.1481 \end{bmatrix}.$$

局部到整体的装配, 写出来就是如下的自然扩展: 对单元 1, 有

$$\tilde{I}_1 = \begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 3.111 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0.0185 \\ 0.0370 \\ 0 \\ 0 \end{bmatrix},$$

对单元 2, 有

$$\tilde{I}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 3.111 & -2.9444 & 0 \\ 0 & -2.9444 & 3.111 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0.0741 \\ 0.0926 \\ 0 \end{bmatrix},$$

对单元 3, 有

$$\tilde{I}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3.111 & -2.9444 \\ 0 & 0 & -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0.1296 \\ 0.1481 \end{bmatrix}.$$

三个扩展相加即得需要的整体刚度矩阵和载荷向量

$$\begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & -2.9444 & 3.1111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0.0185 \\ 0.1111 \\ 0.2222 \\ 0.1481 \end{bmatrix}.$$

式 (4.5) 的右端还有积分产生的边界项 $u'v|_0^1$, 现在考察它的贡献. 第 j 个方程是取 $v = \Phi_j$, 因此该方程的右端要加上

$$\begin{aligned} v = \Phi_j : \quad u'v|_0^1 &= u'\Phi_j|_0^1 = u'(1)\Phi_j(x_n) - u'(0)\Phi_j(x_0) \\ &= u'(1)\delta_{jn} - u'(0)\delta_{j0}. \end{aligned}$$

显然只需要在第 0 行加上 $-u'(0)$, 最后一行加上 $u'(1)$, 最终得到

$$\begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & -2.9444 & 3.1111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0.0185 - u'(0) \\ 0.1111 \\ 0.2222 \\ 0.1481 + u'(1) \end{bmatrix}. \quad (4.6)$$

最后考虑边界条件的处理. 若给出的是 Dirichlet 边界条件 $u(0) = u(1) = 0$, 则理论上我们选择的检验函数空间要求 $v(0) = v(1) = 0$, 从而 (4.5) 中的 $u'v|_0^1$ 自动消除. 对非齐次情形可通过边界条件齐次化转化为齐次情形, 但此时方程会发生变化, 变分形式相应地有所改变. 我们真正处理时并不是这样做的, 而是把第一行和最后一行分别用恒等式 $u_0 = u(0)$ 和 $u_3 = u(1)$ 代替, 对这里的例子就是

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u(0) \\ 0.1111 \\ 0.2222 \\ u(1) \end{bmatrix}.$$

这种做法有一个缺点, 就是破坏了矩阵的对称性. 为此可以把已知节点值移到右端

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u(0) \\ 0.1111 + 2.9444u_0 \\ 0.2222 + 2.9444u_3 \\ u(1) \end{bmatrix},$$

其中 $u_0 = u(0)$, $u_3 = u(1)$. 为了降低矩阵的规模, 可考虑去除恒等的行, 即

$$\begin{bmatrix} 6.2222 & -2.9444 \\ -2.9444 & 6.2222 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0.1111 + 2.9444u_0 \\ 0.2222 + 2.9444u_3 \end{bmatrix}.$$

以后都采用这种处理. 类似可处理其他边界条件, 例如, 当左边界是 Dirichlet 边界条件, 而右边界是 Neumann 边界条件时, 则只要把 (4.6) 的第一行恒等替换, 而最后一行代入 Neumann 边界值即可.

注 4.1 以后规定: 先处理 Neumann 边界条件, 最后处理 Dirichlet 边界条件. 这一点对高维问题比较重要. 例如对矩形区域, 设上边界为 Neumann, 右边界为 Dirichlet. 若先处理 Dirichlet, 再处理 Neumann, 则右上角的点可能变成未知点 (除非人为记住, 这样没有前者方便).

为了程序的统一, 我们在主程序中如下给定边界条件信息.

```
1 bdStruct = setboundary1(node, elem);
```

它含有 Neumann 和 Dirichlet 边界点的序号.

Neumann 边界条件如下处理.

```
1 %% Assemble Neumann boundary conditions
2 Neumann = bdStruct.Neumann;
3 if ~isempty(Neumann)
4     nvec = 1;
5     if find(elem(:,1)==Neumann), nvec = -1; end
6     Dnu = pde.Du(node(Neumann,:))*nvec;
7     ff(Neumann) = ff(Neumann) + a*Dnu;
8 end
```

注意, 当 Neumann 为一维单元的左端点时, 外法向量为 -1, 否则为 1.

Dirichlet 边界比较容易, 用恒等式法替换.

```
1 % ----- Dirichlet boundary conditions -----
2 Dirichlet = bdStruct.Dirichlet; g_D = pde.g_D;
3 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
4 bdNode = (isBdNode); freeNode = (~isBdNode);
5 u = zeros(N,1); u(bdNode) = g_D(node(Dirichlet));
6 ff = ff - kk*u;
```

4.3.4 程序整理

函数文件

函数文件的组成部分在前面已经给出, 具体可参见 GitHub. 主程序如下.

```
1 %% Parameters
2 maxIt = 5;
3 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
4 ErrL2 = zeros(maxIt,1);
5 ErrH1 = zeros(maxIt,1);
6
7 %% Generate an initial mesh
8 a = 0; b = 1;
9 N = 11; % numbers of nodes
10 node = linspace(a,b,N)';
11 elem = [(1:N-1)', (2:N)'];
12 bdNeumann = 'abs(x-1)<1e-4';
13
14 %% Get the PDE data
15 a = 1; b = 1; c = 0;
16 para = struct('a',a, 'b',b, 'c',c);
17 pde = pde1D(para);
```

```

18
19 %% Finite element method
20 for k = 1:maxIt
21     % refine mesh
22     [node,elem] = uniformrefine1(node,elem);
23     % set boundary
24     bdStruct = setboundary1(node,elem);
25     % solve the equation
26     uh = FEM1D(node,elem,pde,bdStruct);
27     % record
28     NNdof(k) = length(uh);
29     h(k) = 1/size(elem,1);
30     if NNdof(k) < 80 % show solution for small size
31         figure(1);
32         [node1,id] = sort(node);
33         plot(node1,uh(id),'r- ',node1,pde.uexact(node1), ...
34             'k*', 'linewidth',2);
35     end
36     % compute error
37     ErrL2(k) = getL2error1(node,elem,pde.uexact,uh);
38     ErrH1(k) = getH1error1(node,elem,pde.Du,uh);
39 end
40
41 %% Plot convergence rates and display error table
42 figure(2);
43 showrateh(h,ErrL2,ErrH1);
44
45 fprintf('\n');
46 disp('Table: Error')
47 colname = {'#Dof','h','|||u-u_h|||','|u-u_h|_1'};
48 disptable(colname,NNdof,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
49
50 %% Conclusion
51 %
52 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
53 % (2nd order) is observed.

```

这里涉及的误差计算函数后面再说明.

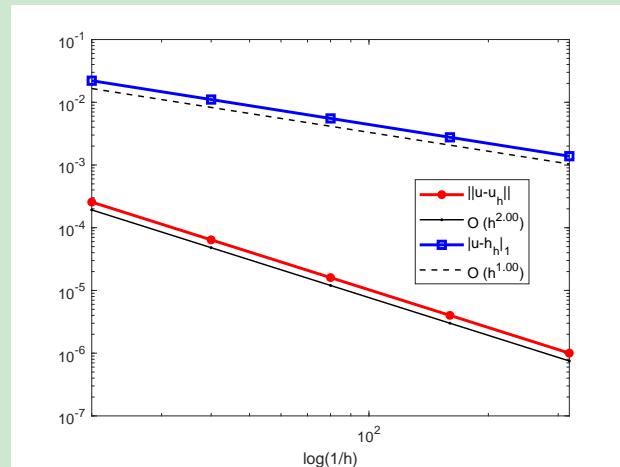


图 4.4. 两点边值问题 P1 元的误差阶

第五章 Poisson 方程的边值问题

从本章开始, 我们把主要精力放在二维问题上. 我们将会发现, 一维问题的处理策略, 可以容易地平移到二维问题上, 这也是本章需要实现的目标.

有限元编程可以归结为三步:

- 对区域进行剖分, 存储需要的网格数据;
- 计算单元刚度矩阵和载荷向量, 并装配;
- 计算代数方程组.

本章只考虑前两步.

5.1 一些说明

5.1.1 问题描述与网格数据

为了简单, 考虑 Poisson 问题

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ \frac{\partial u}{\partial n} = g_N & \text{on } \Gamma_N. \end{cases} \quad (5.1)$$

计算中取精确解为

$$u(x, y) = y^2 \sin(\pi x).$$

区域及剖分如下图所示, 且假设右边界是 Neumann 边界条件, 其他为 Dirichlet 边界条件.

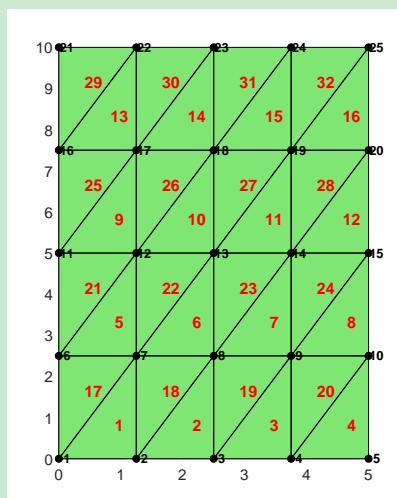


图 5.1. 矩形区域的标准三角剖分 (红色数字为单元编号)

不考虑边界条件, (5.1) 对应的变分形式为

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx dy - \int_{\partial\Omega} v \frac{\partial u}{\partial n} \, ds = \int_{\Omega} f v \, dx dy. \quad (5.2)$$

如同一维问题, 我们最后考虑边界条件和边界积分

$$-\int_{\partial\Omega} v \frac{\partial u}{\partial n} \, ds.$$

注意, 对有限元问题, 检验函数空间 V_h 中的函数总是要求在 Dirichlet 边界处为零, 例如协调元空间为

$$V_h = \{v_h \in C^0(\bar{\Omega}_h) : v_h|_K \in \mathbb{P}_k(K), \quad v_h|_{\Gamma_D} = 0\}.$$

试探函数空间则为

$$V_h^{g_D} = \{v_h \in C^0(\bar{\Omega}_h) : v_h|_K \in \mathbb{P}_k(K), \quad v_h|_{\Gamma_D} = g_D\}.$$

有限元问题为: 找 $u_h \in V_h^{g_D}$ 使得

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, dx dy = \int_{\Omega} f v_h \, dx dy + \int_{\Gamma_N} \partial_n u_h v_h \, ds, \quad v_h \in V_h. \quad (5.3)$$

如图 5.1, 矩形区域用标准三角形进行剖分, 红色数字表示单元编号, 黑色数字表示节点编号. 我们需要给出节点坐标、连通性以及局部整体编号关系等信息, 下面逐一考虑.

节点坐标

在编程中我们需要每个节点的坐标, 用 `node` 记录, 它是两列的一个矩阵, 第一列表示各节点的横坐标, 第二列表示各节点的纵坐标, 行的索引号对应节点的整体编号.

连通性

我们还要给出每个单元连通的顶点, 这里用 `elem` 表示 (每行对应一个单元), 如第 1 个单元连通的节点是 1, 2, 7, 有

```
1 elem(1,1)=1; elem(1,2)=2; elem(1,3)=7; % 第 1 个单元
```

局部整体对应

对每个单元三角形, 局部编号都是 1, 2, 3 (按逆时针方向). 如同一维问题, 在指标编程法中, 我们用 `index` 定位相应的整体编号. 例如, 对单元 ①, 可给出如下的局部整体对应

$$\{1, 2, 3\} \text{ (local)} \rightarrow \{1, 2, 7\} \text{ (global)},$$

这样对该单元有

```
index(1)= 1; index(2)= 2; index(3)= 7;
```

在连通性中我们已经获得了每个单元的顶点标号 (逆时针), 对单元 `iel`, 有

```
index = elem(iel,:);
```

剖分图 5.1 的 `node`, `elem` 使用 `squaremesh.m` 函数生成 (参考 iFEM). 有了 `node` 和 `elem`, 我们就可以画出剖分图 5.1:

```
showmesh(node,elem); findnode(node); findelem(node,elem);
```

节 2.4 给出了边界设置函数 `setboundary.m`, 如下获取 Neumann 和 Dirichlet 边界.

```
1 bdNeumann = 'abs(x-1)<1e-4'; % string for Neumann
2 bdStruct = setboundary(node,elem,bdNeumann);
```

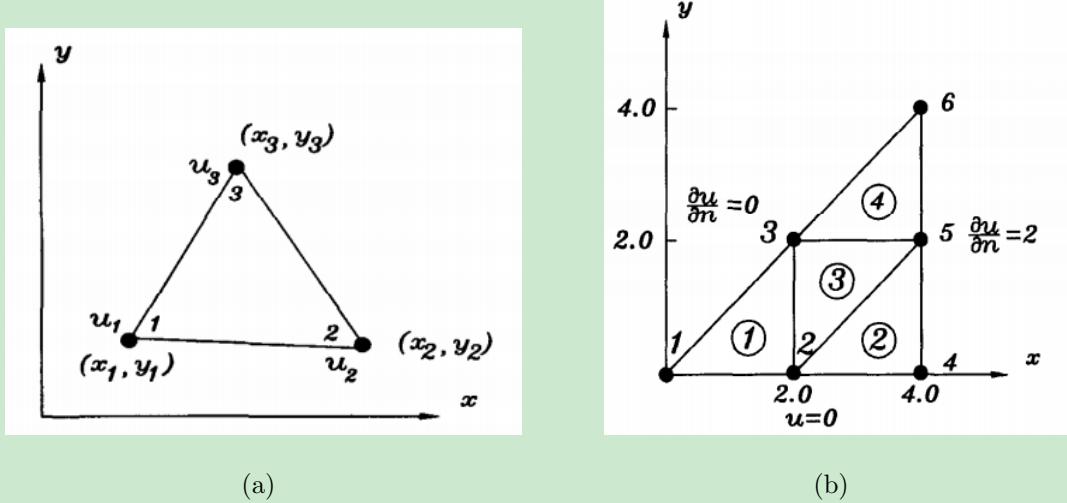
这里,

- `bdStruct` 是结构体, 包含了各种边界信息.
- `bdNodeIdxD` = `bdStruct.bdNodeIdxD` 记录的是不重复的 Dirichlet 边界节点编号.
- `bdEdgeN` = `bdStruct.bdEdgeN` 记录的是 Neumann 边界单元, 即小区间的起点和终点编号.
- Neumann 边界之所以按一维的 `elem` 记录, 是因为 Neumann 条件可视为一维的装配问题, 后面将会看到.

5.1.2 二维问题的装配

指标装配法

节 4.2 给出了一维问题刚度矩阵和载荷向量的装配算法, 其实对二维以及更高维问题, 该算法同样适用. 下面用一个例子来说明.



(a)

(b)

图 5.2. 单元剖分 (右图的圈表示单元编号)

以图 5.2 (b) 的剖分为例说明. 不考虑边界项, 变分形式的左端为

$$a(u, v) = \int_{\Omega} (u_x v_x + u_y v_y) dx dy.$$

为了方便, 考虑线性 Lagrange 有限元. 如图 5.2 (a), 单元上的线性插值为

$$u = \phi_1(x, y)u_1 + \phi_2(x, y)u_2 + \phi_3(x, y)u_3,$$

其中

$$\begin{aligned}\phi_1 &= \frac{1}{2S} [(x_2 y_3 - x_3 y_2) + (y_2 - y_3)x + (x_3 - x_2)y], \\ \phi_2 &= \frac{1}{2S} [(x_3 y_1 - x_1 y_3) + (y_3 - y_1)x + (x_1 - x_3)y], \\ \phi_3 &= \frac{1}{2S} [(x_1 y_2 - x_2 y_1) + (y_1 - y_2)x + (x_2 - x_1)y],\end{aligned}$$

满足

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}, \quad \phi_i(z_j) = \delta_{ij}, \quad \phi_1 + \phi_2 + \phi_3 = 1.$$

注意到线性基函数求导后为常数, 对 $u = \phi_1 u_1 + \phi_2 u_2 + \phi_3 u_3$, 逐个代入 $v = \phi_j$, $j = 1, 2, 3$ 易得单元刚度矩阵为

$$[K^e] = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix},$$

这里

$$k_{11} = \frac{1}{4S} [(x_3 - x_2)^2 + (y_2 - y_3)^2],$$

$$k_{12} = k_{21} = \frac{1}{4S} [(x_3 - x_2)(x_1 - x_3) + (y_2 - y_3)(y_3 - y_1)],$$

$$k_{13} = k_{31} = \frac{1}{4S} [(x_3 - x_2)(x_2 - x_1) + (y_2 - y_3)(y_1 - y_2)],$$

$$k_{22} = \frac{1}{4S} [(x_1 - x_3)^2 + (y_3 - y_1)^2],$$

$$k_{23} = k_{32} = \frac{1}{4S} [(x_1 - x_3)(x_2 - x_1) + (y_3 - y_1)(y_1 - y_2)],$$

$$k_{33} = \frac{1}{4S} [(x_2 - x_1)^2 + (y_1 - y_2)^2].$$

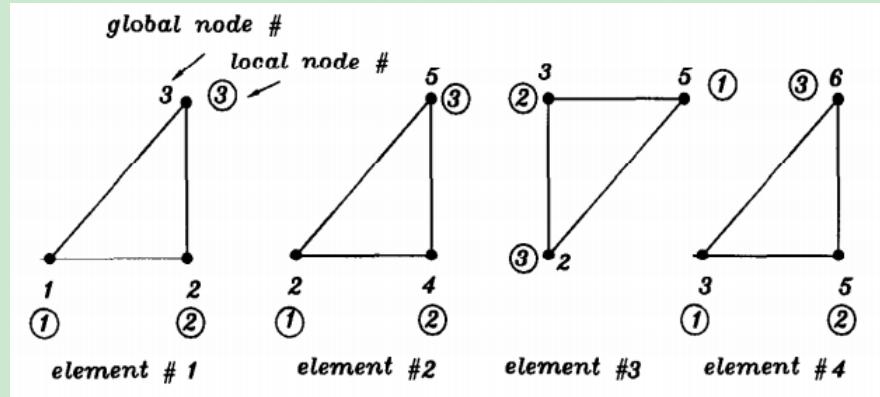


图 5.3. 局部与整体编号 (圈是局部标号)

若图 5.2 (b) 中的四个单元三角形都按图 5.3 进行局部编号 (四个单元的局部编号用带圈数字表示), 则每个单元的刚度矩阵都是

$$[K^e] = \begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix}.$$

按照整体排序, 我们有 (原来在哪行, 还是应该在那行)

1. *element #1*

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.5 & -0.5 & 0.0 & 0 & 0 & 0 \\ -0.5 & 1.0 & -0.5 & 0 & 0 & 0 \\ 0.0 & -0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

2. element #2

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_2 \\ u_4 \\ u_5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & -0.5 & 0.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 1.0 & -0.5 & 0 \\ 0 & 0.0 & 0 & -0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

3. element #3

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_5 \\ u_3 \\ u_2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & -0.5 & 0 & 0.0 & 0 \\ 0 & -0.5 & 1.0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.0 & -0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

4. element #4

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & -0.5 & 0.0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0 & 1.0 & -0.5 \\ 0 & 0 & 0.0 & 0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

相加后我们就把单元刚度矩阵装配成整体刚度矩阵.

可以看到上面的处理与一维问题没有什么区别, 只是扩展时变量个数更多罢了. 显然只要给出局部编号与整体编号的关系向量 `index`, 就可按一维的算法合成整体刚度矩阵及载荷向量, 即如下进行

```

1 N = size(node,1); NT = size(elem,1);
2 kk = zeros(N,N); ff = zeros(N,1);
3 for iel = 1:NT
4     % local --> global
5     index = elem(iel,:);
6     % assemble
7     kk(index,index) = kk(index,index)+ke;
8     ff(index) = ff(index)+fe;
9 end
10 kk = sparse(kk);

```

这里, \mathbf{k}_e 和 \mathbf{f}_e 是单元循环过程中产生的刚度矩阵和载荷向量. 指标装配不是高效的方法, 后面不再给出详细的程序.

sparse 装配法

与 CODE 4.1 一样, 我们可以给出 sparse 装配指标 (把所有自由度想象成一维问题的点), 只要把 $\text{Ndof} = 2$ 改为 $\text{Ndof} = 3$, 即

```
1 ii = reshape(repmat(elem, Ndof, 1), [], 1);
2 jj = repmat(elem(:, ), Ndof, 1);
```

单元刚度矩阵要按照如下方式排列

```
1 K = [k11, k12, k13, k21, k22, k23, k31, k32, k33];
```

即按行拉直每个单元刚度矩阵, 并逐行排列单元, 从而 \mathbf{K} 的第 1 列为所有单元的 k_{11} , 依次类推. 这样, 刚度矩阵如下装配

```
1 ss = K(:, );
2 kk = sparse(ii, jj, ss, N, N);
```

单元载荷向量如下排列

```
1 F = [f1, f2, f3];
```

这里, \mathbf{f}_i 是所有单元的. 如下装配

```
1 ff = accumarray(elem(:, ), F(:, ), [N 1]);
```

5.2 Poisson 方程的一阶有限元方法

5.2.1 刚度矩阵的计算

单元变分形式为

$$a_K(u, v) = \int_K \nabla u \cdot \nabla v \, dx \, dy,$$

设局部节点基为 ϕ_1, ϕ_2, ϕ_3 , 则有

$$A_K = (a_K(\phi_j, \phi_i))_{3 \times 3}.$$

对线性元, $\phi_i(x, y) = \lambda_i(x, y)$ 就是面积坐标函数. 设 i, j, k 表示 1,2,3 的轮换. 定义

$$\xi_i = x_j - x_k, \quad \eta_i = y_j - y_k, \quad \omega_i = x_j y_k - x_k y_j,$$

则三角形的有向面积可表示为

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} = \frac{1}{2} (\xi_1 \eta_2 - \xi_2 \eta_1) = \omega_1 \omega_2 \omega_3,$$

且

$$\lambda_i(x, y) = \frac{1}{2S} (\eta_i x - \xi_i y + \omega_i), \quad i = 1, 2, 3. \quad (5.4)$$

从 (x, y) 到 (λ_1, λ_2) 变换的 Jacobi 矩阵的行列式为

$$|J| = \det \frac{\partial(\lambda_1, \lambda_2)}{\partial(x, y)} = \frac{1}{2S},$$

且有

$$\frac{\partial \lambda_i}{\partial x} = \frac{\eta_i}{2S}, \quad \frac{\partial \lambda_i}{\partial y} = -\frac{\xi_i}{2S}, \quad i = 1, 2, 3.$$

显然 λ_i 的导数是常数, 于是

$$\int_K \nabla \phi_j \cdot \nabla \phi_i dx dy = \nabla \phi_j \cdot \nabla \phi_i \cdot |K|,$$

为此需要计算基函数的导数和面积. 我们用三维数组 $Dphi$ 存储三个基函数的导数, 如下

```

1 function [Dphi,area] = gradbasis(node,elem)
2
3 z1 = node(elem(:,1,:));
4 z2 = node(elem(:,2,:));
5 z3 = node(elem(:,3,:));
6 e1 = z2-z3; e2 = z3-z1; e3 = z1-z2;
7 area = 0.5*(-e3(:,1).*e2(:,2)+e3(:,2).*e2(:,1));
8
9 grad1 = [e1(:,2), -e1(:,1)]./(2*area); % stored in rows
10 grad2 = [e2(:,2), -e2(:,1)]./(2*area);
11 grad3 = -(grad1+grad2);
12
13 NT = size(elem,1);
14 Dphi(1:NT,:,:1) = grad1;
15 Dphi(1:NT,:,:2) = grad2;
16 Dphi(1:NT,:,:3) = grad3;

```

这里, $grad1$ 每行对应一个单元. 刚度矩阵如下一次性计算

```

1 K = zeros(NT,Ndof^2);
2 s = 1;
3 for i = 1:Ndof
4     for j = 1:Ndof
5         K(:,s) = sum(Dphi(:,:,i).*Dphi(:,:,j),2).*area;
6         s = s+1;
7     end
8 end

```

这里, $Dphi(:,:,i) .* Dphi(:,:,j)$ 是横坐标点乘横坐标, 纵坐标点乘纵坐标给出的向量, 按行求和就是 $\nabla\phi_j \cdot \nabla\phi_i$.

注 5.1 较新版本的 MATLAB, 如 MATLAB 2018 支向量与矩阵的点乘 (Python 的 NumPy 称其为广播). 设 $A = (a_{ij})$ 为 $m \times n$ 的矩阵, $b = (b_i)$ 是 m 维的列向量, 则

- $A.*b$ 与 $b.*A$ 相同, 都为 $(b_i a_{ij})_{m \times n}$, 即行之间相乘.
- $A./b$ 为 $(a_{ij}/b_i)_{m \times n}$, $b./A$ 为 $(b_i/a_{ij})_{m \times n}$, 都是行之间相除.
- 也就是说, 向量首先扩展为与 A 同样大小的矩阵, 再进行“点运算”.

这样, 第二个循环也可去掉, 如下

```

1 K = zeros(NT,Ndof^2);
2 for i = 1:Ndof
3     j = 1:Ndof;    jd = (i-1)*Ndof+1:i*Ndof;
4     K(:,jd) = sum(Dphi(:,:,i).*Dphi(:,:,j),2).*area;
5 end
6 kk = sparse(ii,jj,K(:,N,N));

```

不过, 在后面的程序中我们还是尽量避免这种运算 (所有程序确保在 MATLAB 2015 上可执行).

5.2.2 载荷向量的计算

载荷向量的积分可用中心格式近似

$$F_K = \int_K \begin{bmatrix} f\phi_1 \\ f\phi_2 \\ f\phi_3 \end{bmatrix} dx dy \approx \begin{bmatrix} f\lambda_1 \\ f\lambda_2 \\ f\lambda_3 \end{bmatrix}_{(x_c, y_c)} \cdot |K| = f(x_c, y_c) \cdot \frac{|K|}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

其中 $f(x_c, y_c)$ 是在单元重心处的值. 如下计算载荷向量

```

1 % coordinates of all triangles
2 z1 = node(elem(:,1),:);
3 z2 = node(elem(:,2),:);
4 z3 = node(elem(:,3),:);
5 % load vector
6 zc = 1/3*(z1+z2+z3);
7 f1 = f(zc).*area./3; f2 = f1; f3 = f1;
8 F = [f1,f2,f3];
9 ff = accumarray(elem(:,1), F(:,1), [N 1]);

```

这里数据文件给出了函数 f , 它是按整体坐标给的, 即 $f = f(p)$, $p = (x, y)$.

也可用三角形上的 Gauss 求积公式. iFEM 中提供了三角形上的 Gauss 求积节点与权重, 即 quadpts.m. 我们简单说明一下其用法. 实际上, 那里的权重和节点是针对面积坐标下的参考三角形 \hat{T} 进行的, 积分公式为

$$\iint_{\hat{T}} f(\lambda_1, \lambda_2, \lambda_3) d\hat{\sigma} \approx |\hat{T}| \sum_{p=1}^{n_g} w_p f(\lambda_{1,p}, \lambda_{2,p}, \lambda_{3,p}), \quad |\hat{T}| = \frac{1}{2},$$

其中,

$$\begin{cases} x = x_1\lambda_1 + x_2\lambda_2 + x_3\lambda_3 \\ y = y_1\lambda_1 + y_2\lambda_2 + y_3\lambda_3 \end{cases}, \quad \lambda_1 + \lambda_2 + \lambda_3 = 1, \quad (5.5)$$

注意到

$$\det \left(\frac{\partial(x, y)}{\partial(\lambda_1, \lambda_2)} \right) = 2S,$$

这里 S 是三角形 T 的代数面积, 我们有

$$\int_T F(x, y) d\sigma = 2|T| \int_{\hat{T}} f(\lambda_1, \lambda_2, \lambda_3) d\hat{\sigma} = |T| \sum_{p=1}^{n_g} w_p f(\lambda_{1,p}, \lambda_{2,p}, \lambda_{3,p}).$$

因变换前后点处的值不变, 故

$$\int_T F(x, y) d\sigma = |T| \sum_{p=1}^{n_g} w_p F(x_p, y_p). \quad (5.6)$$

利用 (5.5) 可把参考元上的 Gauss 点 $(\lambda_{1,p}, \lambda_{2,p}, \lambda_{3,p})$ 转化为 T 上的点.

```

1 % n: n-th order quadrature rule
2 f = pde.f;
3 [lambda,weight] = quadpts(2); % n = 2
4 f1 = zeros(NT,1); f2 = f1; f3 = f1;
5 for iel = 1:NT
6     vK = node(elem(iel,:)); % vertices of K
7     pxy = lambda*vK; fxy = f(pxy);
8     fv1 = fxy.*lambda(:,1); % (f,phi1)
9     fv2 = fxy.*lambda(:,2); % (f,phi2)
10    fv3 = fxy.*lambda(:,3); % (f,phi3)
11
12    f1(iel,:) = area(iel)*weight*fv1;
13    f2(iel,:) = area(iel)*weight*fv2;
14    f3(iel,:) = area(iel)*weight*fv3;
15 end
16 F = [f1,f2,f3];
17 ff = accumarray(elem(:,1), F(:,1), [N 1]);

```

注意, `lambda` 共有 3 列, 第 i 列对应 λ_i , 该列数组就是对应的积分点, 而 `weight` 是行向量. 也可直接用向量化运算

```

1 % Gauss quadrature rule
2 [lambda,weight] = quadpts(2);
3 F = zeros(NT,3);
4 for iel = 1:NT
5     vK = node(elem(iel,:,:) ); % vertices of K
6     pxy = lambda*vK; fxy = f(pxy);
7     fv = fxy.*lambda;
8     F(iel,:) = area(iel)*weight*fv;
9 end
10 ff = accumarray(elem(:, ), F(:, ),[N 1]);

```

上面可继续优化. 有限元计算过程中要尽量避免单元循环, 而以向量化代替. 前面是把 (5.6) 的和向量化运算, 但更理想的方式是先计算出求和中的每一项, 再把和项相加. 为此, 我们要给出所有单元的 pxy. 第 p 个和项的所有 pxy 如下

```

1 % quadrature points in the x-y coordinate
2 pxy = lambda(p,1)*node(elem(:,1),:) ...
3     + lambda(p,2)*node(elem(:,2),:) ...
4     + lambda(p,3)*node(elem(:,3),:) ;

```

从而积分如下计算

```

1 F = zeros(NT,3);
2 for p = 1:size(lambda,1)
3     % quadrature points in the x-y coordinate
4     pxy = lambda(p,1)*node(elem(:,1),:) ...
5         + lambda(p,2)*node(elem(:,2),:) ...
6         + lambda(p,3)*node(elem(:,3),:) ;
7     fxy = f(pxy); fv = fxy*lambda(p,:);
8     F = F + weight(p)*fv;
9 end
10 F = area.*F;
11 ff = accumarray(elem(:, ), F(:, ),[N 1]);

```

注 5.2 需要注意的是, 对常函数 $f(x, y) = 1$, 调用匿名函数 $f = @(\mathbf{x}, \mathbf{y}) 1$ 时, 多个点只会生成一个值, 此时必须改为 $f = @(\mathbf{x}, \mathbf{y}) 1 * ones(\text{size}(\mathbf{x}))$ 或 $f = @(\mathbf{x}, \mathbf{y}) 1 + 0 * \mathbf{x}$ (编写函数时也要注意).

5.2.3 边界条件的处理

先考虑 Neumann 边界条件, 即 (5.3) 中的边界积分项.

等效拉平法

设区域 Ω 剖分后所得区域为 Ω_h , 变分问题实际上是在 Ω_h 上考虑.

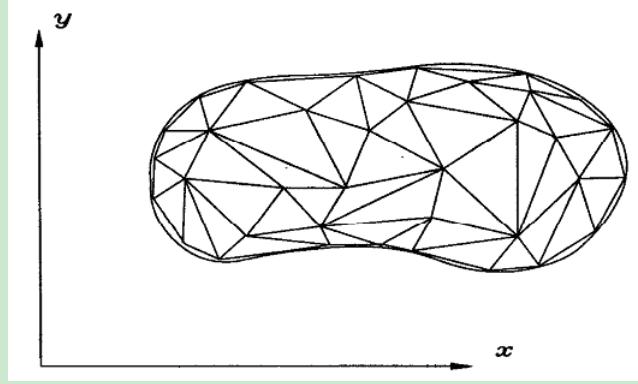


图 5.4. 区域剖分

现在考虑 (5.3) 中的边界积分

$$\int_{\Gamma_N} \frac{\partial u}{\partial n} v ds.$$

设 $\Gamma_N = e_1 \cup e_2 \cup \dots \cup e_l$, 则

$$\int_{\Gamma_N} \frac{\partial u}{\partial n} v ds = \sum_i \int_{e_i} \frac{\partial u}{\partial n} v ds, \quad (5.7)$$

这里边界项 $\int_{e_i} \frac{\partial u}{\partial n} v ds$ 相当于一维问题的 $u'v|_{x_{i-1}}^{x_i}$, 而 $\int_{\partial\Omega} \frac{\partial u}{\partial n} v ds$ 相当于 $u'v|_0^1$ (见式 (4.5)).

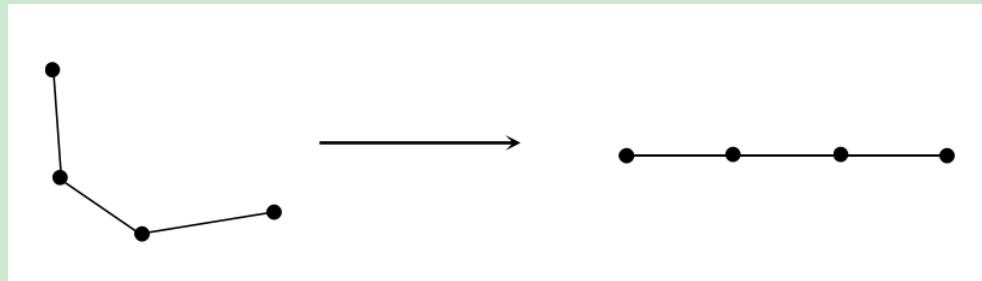


图 5.5. 边界“拉平”

值得注意的是, 若把 $\partial\Omega_h$ “拉平” (想象成一条直线), 则边界上的积分就相当于一维问题的有限元积分, 这是我们处理高维问题的重要观察. 事实上, 设整体变量排列为 u_1, u_2, \dots, u_M , 则有相应的整体节点基为 $\Phi_1, \Phi_2, \dots, \Phi_M$, 满足 $\Phi_j(x_i) = \delta_{ij}$ (x_i 是整体节点). 不妨设前 l 个为全部边界点, 则

$$\Psi_j = \Phi_j|_{\partial\Omega_h}, \quad j = 1, 2, \dots, l$$

相当于“拉平”边界对应的整体节点基 (一维问题), 因为它满足节点基的定义. 而 Ψ_j 限制在边界单元上就会产生类似一维问题的局部节点基 ϕ_1, ϕ_2 .

这样, 对 (5.7) 就可以类似一维问题那样分单元装配载荷向量.

5.2.4 边界积分的装配

方程组右端的第 j 行要加上

$$v = \Phi_j : \int_{\Gamma_N} \frac{\partial u}{\partial n} v ds = \sum_i \int_{e_i} \frac{\partial u}{\partial n} v ds,$$

前面已经分析了它可以类似一维问题处理, 下面具体讨论之.

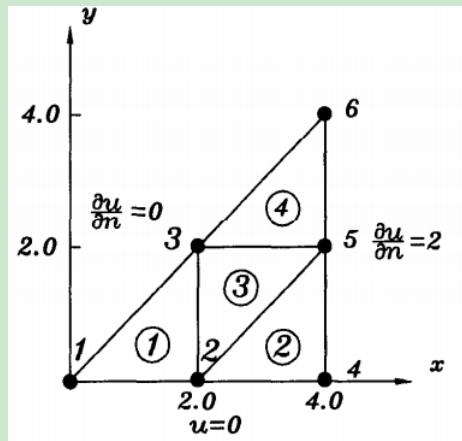


图 5.6. 单元剖分 (圈表示单元编号)

为了方便, 我们用图 5.6 的区域剖分来说明边界积分的影响, 这里边界条件包含 Dirichlet 边界条件和 Neumann 边界条件.

- 先看 Neumann 边界 4-5, 记为 Γ^{45} , 计算 $\int_{\Gamma^{45}} \frac{\partial u}{\partial n} v ds$.

此时通量 $\frac{\partial u}{\partial n} = g_N$ 已知, 从而积分可以具体算出, 需要把算出的值放到方程的右端 (载荷向量中). 单元为 $[x_4, x_5]$, 我们有局部整体对应

$$\{1, 2\} \text{ (local)} \quad \rightarrow \quad \{4, 5\} \text{ (global)},$$

从而单元积分应贡献的行如下

$$\begin{bmatrix} \int_{\Gamma^{45}} g_N \phi_1 ds \\ \int_{\Gamma^{45}} g_N \phi_2 ds \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} F_4 \\ F_5 \end{bmatrix},$$

其中 ϕ_1, ϕ_2 是局部节点基. 令指标

$$\text{index}(1) = 4; \quad \text{index}(2) = 5;$$

则可利用载荷向量的装配算法计算. 对 Γ^{56} 类似处理.

- 再看 Dirichlet 边界 1-2, 记为 Γ^{12} .

u_1, u_2 的值是已知的, 方程组对应的行最后都用恒等式 $u_1 = u(x_1)$ 和 $u_2 = u(x_2)$ 替换.

根据以上分析, 可以给出如下的编程策略.

二维问题的有限元编程策略

- (1) 不考虑边界条件或边界积分项, 按 5.1.2 节求出暂时的整体刚度矩阵和整体载荷向量.
 - (2) 确定 Neumann 边界, 如图 5.6 的 4-5-6 部分, 把这部分边界视为一维问题的整体区域, $\frac{\partial u}{\partial n} = g_N$ 视为一维问题的载荷 f , 从而给出“边界单元载荷向量”, 然后加到整体载荷向量的对应位置上 (若 $g_N = 0$, 则贡献为 0, 直接忽略).
 - (3) 确定 Dirichlet 边界, 用“恒等式法”取代对应边界点的系统方程的行 (一般去除 Dirichlet 节点“变量”).
-

注 5.3 必须等所有装配过程完成后才能应用 Dirichlet 边界条件, 而 Neumann 边界条件本身是装配问题, 所以必须先处理 Neumann 边界条件, 最后处理 Dirichlet 边界条件.

注 5.4 Neumann 边界对应边界积分, 它可以逐段积分, 而且与积分单元的次序无关, 只要保证每个单元的定向正确即可.

边界单元的积分计算

我们用梯形公式近似 (或用中心格式)

$$\int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_1 ds = \frac{h_e}{2} \left(\frac{\partial u}{\partial n}(z_1) \phi_1(z_1) + \frac{\partial u}{\partial n}(z_2) \phi_1(z_2) \right) = \frac{h_e}{2} \frac{\partial u}{\partial n}(z_1),$$

$$\int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_2 ds = \frac{h_e}{2} \left(\frac{\partial u}{\partial n}(z_1) \phi_2(z_1) + \frac{\partial u}{\partial n}(z_2) \phi_2(z_2) \right) = \frac{h_e}{2} \frac{\partial u}{\partial n}(z_2),$$

其中

$$h_e = |z_j - z_i| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

注意 h_e 是单元长度而不是代数长度, 这是因为边界积分可以看成拉平后的一维积分, 因是逆时针排列, 故拉平后是从左到右, 因而是单元长度.

注意到

$$h_e \frac{\partial u}{\partial n} = h_e \nabla u \cdot \vec{n} = \nabla u \cdot (h_e \vec{n}),$$

而 $\hat{n}_e = h_e \vec{n}$ 可通过三角形的定向边逆时针旋转 90° 获得 (绕着定向边的终点), 即

```

1 z1 = node(bdEdgeN(:,1),:); z2 = node(bdEdgeN(:,2),:);
2 e = z1-z2; % e = z2-z1
3 ne = [-e(:,2),e(:,1)]; % scaled ne

```

注意, setboundary.m 函数给出的边界边已经定向过.

边界条件的程序实现

Neumann 边界条件可视为一维问题载荷向量的装配, 为此要给出 Neumann 边界单元“拉平”后从左至右的节点编号 (这里的从左到右意思是按逆时针方向给出的). 前面已经给出, 为 elemN.

Neumann 边界条件可如下快速装配

```
1 %% Assemble Neumann boundary conditions
2 bdEdgeN = bdStruct.bdEdgeN;
3 if ~isempty(bdEdgeN)
4     z1 = node(bdEdgeN(:,1),:); z2 = node(bdEdgeN(:,2),:);
5     e = z1-z2; % e = z2-z1
6     ne = [-e(:,2),e(:,1)]; % scaled ne
7     Du = pde.Du;
8     gradu1 = Du(z1); gradu2 = Du(z2);
9     F1 = sum(ne.*gradu1,2)./2; F2 = sum(ne.*gradu2,2)./2;
10    FN = [F1,F2];
11    ff = ff + accumarray(bdEdgeN(:,1), FN(:,1), [N 1]);
12 end
```

我们总假设有 Dirichlet 边界条件, 类似一维问题处理

```
1 %% Apply Dirichlet boundary conditions
2 bdNodeIdxD = bdStruct.bdNodeIdxD; g_D = pde.g_D;
3 isBdNode = false(N,1); isBdNode(bdNodeIdxD) = true;
4 bdDof = (isBdNode); freeDof = (~isBdNode);
5 nodeD = node(bdDof,:);
6 u = zeros(N,1); u(bdDof) = g_D(nodeD);
7 ff = ff - kk*u;
```

最后的方程组可如下求解

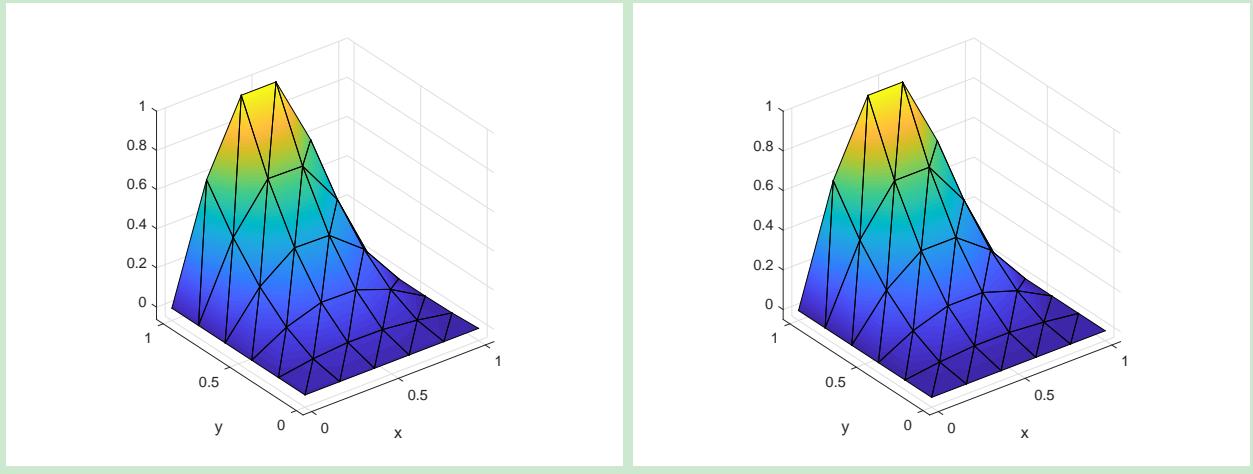
```
1 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
```

MATLAB 早期版本, 例如 MATLAB R2013a 可以用如下语句画图

```
1 p = node'; t = elem';
2 figure, pdesurf(p,t,u);
```

内部主要使用 pdeplot 画图的. 新版本有所改动, 已不支持上面的方式, 这是因为, MATLAB 网格数据中连通性 t 还有额外的信息. 前面给出了 showsolution.m, 它使用 patch 画图, 用法如下:

```
figure, showsolution(node,elem,u);
```



(a) 数值解

(b) 精确解

图 5.7. 模型问题的数值解与精确解

为了便于对比, 这里对画图的区域做了界定, 特别是 z 轴方向. 如果想去掉这些界定, 例如画绝对误差图, 那么可在画图语句后添加 `zlim('auto')`, 如下图

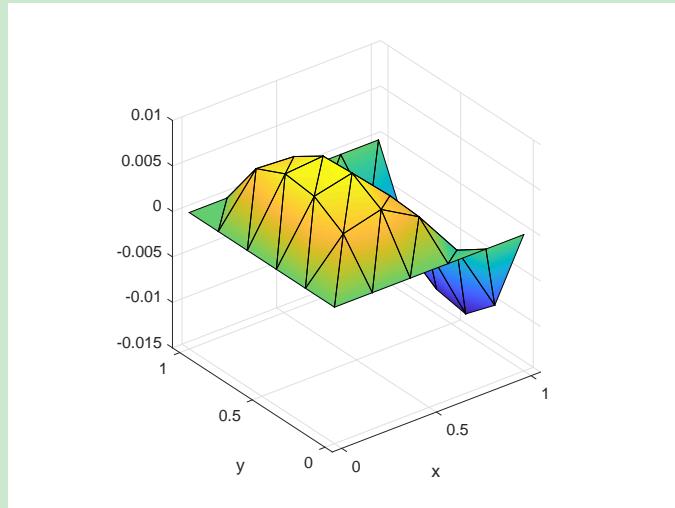


图 5.8. 绝对误差

定义整体相对误差

$$\text{Err} = \frac{\|U - u\|_2}{\|u\|_2},$$

下表给出不同划分的结果.

表 5.1. 整体相对误差 (M 是横向划分, N 是纵向划分)

M	$N = 10$	$N = 20$	$N = 50$
5	1.3931e-02	1.2994e-02	1.2098e-02
10	3.9026e-03	3.3002e-03	2.8684e-03
20	2.2352e-03	9.8932e-04	7.5369e-04

5.2.5 程序整理

具体函数文件见 GitHub. 主程序如下

```

1 clc; clear; close all;
2 %% Parameters
3 maxIt = 5;
4 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
5 ErrL2 = zeros(maxIt,1);
6 ErrH1 = zeros(maxIt,1);
7
8 %% Generate an initial mesh
9 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
10 Nx = 4; Ny = 4; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
11 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
12 bdNeumann = 'abs(x-1)<1e-4';
13
14 %% Get the PDE data
15 pde = Poissondata();
16
17 %% Finite element method
18 for k = 1:maxIt
19     % refine mesh
20     [node,elem] = uniformrefine(node,elem);
21     % set boundary
22     bdStruct = setboundary(node,elem,bdNeumann);
23     % solve the equation
24     uh = Poisson(node,elem,pde,bdStruct);
25     % record
26     NNdof(k) = length(uh);
27     h(k) = 1/(sqrt(size(node,1))-1);
28     if NNdof(k)<2e3
29         figure(1);
30         showresult(node,elem,pde.uexact,uh);
31     end
32     % compute error
33     ErrL2(k) = getL2error(node,elem,pde.uexact,uh);
34     ErrH1(k) = getH1error(node,elem,pde.Du,uh);
35 end
36

```

```

37 %% Plot convergence rates and display error table
38 figure(2);
39 showrateh(h,ErrL2,ErrH1);
40
41 fprintf('\n');
42 disp('Table: Error')
43 colname = {'#Dof','h','||u-u_h|| ','|u-u_h|_1'};
44 disptable(colname,NNdof,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
45
46 %% Conclusion
47 %
48 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
49 % (2nd order) is observed.

```

注 5.5 我们始终假设有 Dirichlet 边界条件.

5.2.6 误差分析

设 u 为精确解, u_h 为数值解, L^2 误差为

$$\text{ErrL2} = \left(\sum_{K \in \mathcal{T}_h} \|u - u_h\|_{0,K}^2 \right)^{1/2},$$

而

$$\begin{aligned} \|u - u_h\|_{0,K}^2 &= \int_K [u - (u_{h1}\phi_1 + u_{h2}\phi_2 + u_{h3}\phi_3)]^2 d\sigma \\ &= |K| \sum_{p=1}^{n_g} w_p [u(z_p) - (u_{h1}\phi_1(z_p) + u_{h2}\phi_2(z_p) + u_{h3}\phi_3(z_p))]^2. \end{aligned}$$

设 `elem2dof` 按单元记录自由度的编号信息, 对 P1-Lagrange 元, 显然有 `elem2dof ... = elem`. 对固定的 p , $u_{h1}\phi_1(z_p) + u_{h2}\phi_2(z_p) + u_{h3}\phi_3(z_p)$ 在所有单元上的结果为

```

1 phi = lambda;
2 uhp = uh(elem2dof(:,1))*phi(p,1) + ...
3     uh(elem2dof(:,2))*phi(p,2) + ...
4     uh(elem2dof(:,3))*phi(p,3);

```

未乘以面积的所有单元误差为

```

1 err = zeros(NT,1);
2 for p = 1:ng
3     % P1 piecewise linear function
4     uhp = uh(elem2dof(:,1))*phi(p,1) + ...
5         uh(elem2dof(:,2))*phi(p,2) + ...
6         uh(elem2dof(:,3))*phi(p,3);
7     % quadrature points in the x-y coordinate
8     pz = lambda(p,1)*node(elem(:,1),:) ...
9         + lambda(p,2)*node(elem(:,2),:) ...

```

```

10      + lambda(p,3)*node(elem(:,3),:);
11  err = err + weight(p)*(uexact(pz) - uhp).^2;
12 end

```

综上, L^2 误差的计算可编写为如下函数.

```

1 function err = getL2error(node, elem, u, uh, quadOrder)
2
3 if nargin == 4, quadOrder = 3; end
4
5 NT = size(elem,1);
6 % Gauss quadrature rule
7 [lambda, weight] = quadpts(quadOrder); ng = length(weight);
8 % elementwise d.o.f.s
9 elem2dof = elem;
10 % area of triangles
11 ve2 = node(elem(:,1),:)-node(elem(:,3),:);
12 ve3 = node(elem(:,2),:)-node(elem(:,1),:);
13 area = 0.5*abs(-ve3(:,1).*ve2(:,2)+ve3(:,2).*ve2(:,1));
14 % basis functions
15 phi = lambda;
16
17 % elementwise error
18 err = zeros(NT,1);
19 for p = 1:ng
20     % P1 piecewise linear function
21     uhp = uh(elem2dof(:,1))*phi(p,1) + ...
22             uh(elem2dof(:,2))*phi(p,2) + ...
23             uh(elem2dof(:,3))*phi(p,3);
24     % quadrature points in the x-y coordinate
25     pz = lambda(p,1)*node(elem(:,1),:);
26             + lambda(p,2)*node(elem(:,2),:);
27             + lambda(p,3)*node(elem(:,3),:);
28     err = err + weight(p)*(pde.uexact(pz) - uhp).^2;
29 end
30 err = area.*err;
31 % Modification
32 err(isnan(err)) = 0; % singular values, i.e. uexact(p) = infy, are excluded
33 err = sqrt(abs(sum(err)));

```

类似可编写计算 H^1 误差的程序.

现在来绘制误差阶的图像. 计算中步长 h 取为单元平均面积的算术平方根, 即 $h = \sqrt{|\Omega|/NT}$. 它正比于 $NT^{-1/2}$, 为此不妨直接取 $h = NT^{-1/2}$. 设 $e = ch^r$, 则误差阶为

$$\log e = r \log h + \log c.$$

为此, 我们用一次多项式拟合 $(\log h, \log e)$, 斜率即一次项的系数为误差阶 r .

```
1 figure,
```

```

2 err = ErrH1;
3 err(err == 0) = 1e-16; % Prevent the case err = 0, log(err) = -Inf.
4 p = polyfit(log(h(1:end)), log(err(1:end)), 1);
5 r = p(1);

```

$(\log h, \log e)$ 的图像可直接用 loglog 函数绘图.

```

1 loglog(h,err,'-*','linewidth',2);

```

为了显示误差阶, 我们在该数值曲线附近画一条以 r 为斜率的直线. 实现在附近的策略是, 保证第一个对应点靠近即可. 数值解的第一个纵坐标为 e_1 , 设所需曲线的纵坐标为 sh^r , 则要求

$$e_1 \sim s_1 h_1^r, \quad s_1 \sim \frac{e_1}{h_1^r}.$$

为此, 可取

$$s_1 = \frac{3}{4} \frac{e_1}{h_1^r}, \quad s(h) = \frac{3}{4} \frac{e_1}{h^r}.$$

画误差阶的函数如下

```

1 function r = showrate(h,err,opt1,opt2)
2
3 err(err == 0) = 1e-16; % Prevent the case err = 0, log(err) = -Inf.
4 p = polyfit(log(h(1:end)), log(err(1:end)), 1);
5 r = p(1);
6 s = 0.75*err(1)/h(1)^r;
7
8 loglog(h,err,opt1,'linewidth',2);
9 hold on
10 loglog(h,s*h.^r,opt2,'linewidth',1);

```

这里, opt1 和 opt2 控制线的颜色、线型等.

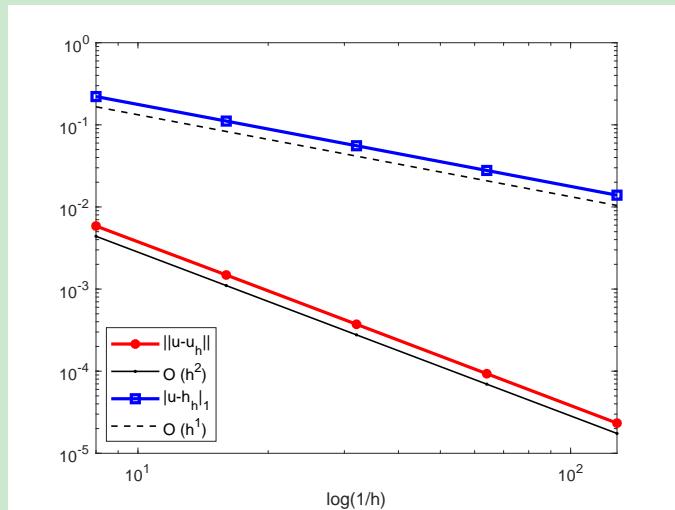


图 5.9. Poisson 方程一阶有限元的 L^2 和 H^1 误差阶

5.3 Poisson 方程的二阶有限元方法

二次 Lagrange 元的局部自由度排列如下图

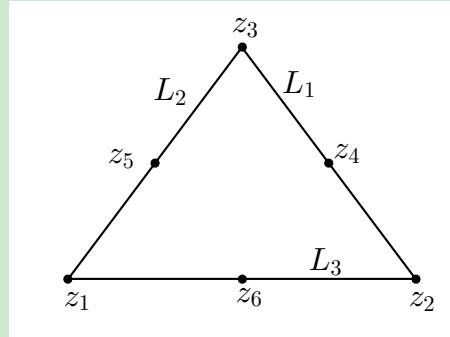


图 5.10. 二次 Lagrange 三角形 (quadratic Lagrange triangle)

整体自由度先排列网格节点值, 接着排列边中点值. 节点基为

$$\phi_i = \lambda_i(2\lambda_i - 1), \quad i = 1, 2, 3,$$

$$\phi_4 = 4\lambda_2\lambda_3, \quad \phi_5 = 4\lambda_3\lambda_1, \quad \phi_6 = 4\lambda_1\lambda_2.$$

稀疏装配指标如下

```

1 %% Get elem2dof
2 % auxstructure
3 auxT = auxstructure(node,elem);
4 edge = auxT.edge;
5 elem2edge = auxT.elem2edge;
6 % numbers
7 N = size(node,1); NT = size(elem,1); NE = size(edge,1);
8 NNdof = N + NE; Ndof = 6; %global and local d.o.f. numbers
9 % elem2dof
10 elem2dof = [elem, elem2edge + N];
11 ii = reshape(repmat(elem2dof, Ndof,1), [], 1);
12 jj = repmat(elem2dof(:, ), Ndof, 1);

```

注意, 这里的 elem2edge 为按单元存储的边的自然序号.

刚度矩阵和载荷向量类似一次 Lagrange 元计算, 只需要修改基函数部分.

```

1 %% Assemble stiffness matrix
2 % lambda and Dlambda
3 quadOrder = 4;
4 [lambda, weight] = quadpts(quadOrder); nG = length(weight);
5 [Dlambda, area] = gradbasis(node,elem);
6 % stiffness matrix
7 K = zeros(NT,Ndof^2); % straighten
8 for p = 1:nG

```

```

9    % Dphi at quadrature points
10   Dphip(:,:,6) = 4*(lambda(p,1)*Dlambda(:,:,2)+lambda(p,2)*Dlambda(:,:,1));
11   Dphip(:,:,1) = (4*lambda(p,1)-1).*Dlambda(:,:,1);
12   Dphip(:,:,2) = (4*lambda(p,2)-1).*Dlambda(:,:,2);
13   Dphip(:,:,3) = (4*lambda(p,3)-1).*Dlambda(:,:,3);
14   Dphip(:,:,4) = 4*(lambda(p,2)*Dlambda(:,:,3)+lambda(p,3)*Dlambda(:,:,2));
15   Dphip(:,:,5) = 4*(lambda(p,3)*Dlambda(:,:,1)+lambda(p,1)*Dlambda(:,:,3));
16   s = 1;
17   for i = 1:Ndof
18       for j = 1:Ndof
19           K(:,s) = K(:,s) + weight(p)*sum(Dphip(:,:,i).*Dphip(:,:,j),2).*area;
20           s = s+1;
21   end
22 end
23 end
24 ii = repmat(elem2dof, Ndof,1), [], 1);
25 jj = repmat(elem2dof(:), Ndof, 1);
26 kk = sparse(ii,jj,K(:,NNdof,NNdof);
27
28 %% Assemble load vector
29 % basis functions
30 phi(:,6) = 4*lambda(:,1).*lambda(:,2);
31 phi(:,1) = lambda(:,1).*(2*lambda(:,1)-1);
32 phi(:,2) = lambda(:,2).*(2*lambda(:,2)-1);
33 phi(:,3) = lambda(:,3).*(2*lambda(:,3)-1);
34 phi(:,4) = 4*lambda(:,2).*lambda(:,3);
35 phi(:,5) = 4*lambda(:,3).*lambda(:,1);
36 % load vector
37 F = zeros(NT,Ndof); % straighten
38 for p = 1:nG
39     % quadrature points in the x-y coordinate
40     pxy = lambda(p,1)*node(elem(:,1),:) ...
41         + lambda(p,2)*node(elem(:,2),:) ...
42         + lambda(p,3)*node(elem(:,3),:);
43     F = F + weight(p)*pde.f(pxy)*phi(p,:);
44 end
45 F = repmat(area,1,Ndof).*F; % F = area.*F;
46 ff = accumarray(elem2dof(:), F(:,[NNdof 1]));

```

Neumann 条件是一维问题的 P2-Lagrange 元, 尽管前面没有介绍, 但与二维问题类似. 局部自由度为一维单元的左右顶点值和中点值. Neumann 条件如下计算和装配.

```

1 %% Assemble Neumann boundary conditions
2 bdEdgeIdxN = bdStruct.bdEdgeIdxN; bdEdgeN = bdStruct.bdEdgeN;
3 if ~isempty(bdEdgeN)
4     % Sparse assembling index
5     elem1 = [bdEdgeN, bdEdgeIdxN + N]; ndof = 3;
6     % Gauss quadrature rule
7     [lambda,weight] = quadpts1(quadOrder); ng = length(weight);
8     % basis function

```

```

9     phi1(:,3) = 4*lambda(:,1).*lambda(:,2);
10    phi1(:,1) = lambda(:,1).*(2*lambda(:,1)-1);
11    phi1(:,2) = lambda(:,2).*(2*lambda(:,2)-1);
12    % nvec
13    z1 = node(bdEdgeN(:,1),:); z2 = node(bdEdgeN(:,2),:); nel = size(bdEdgeN,1);
14    e = z1-z2; he = sqrt(sum(e.^2,2));
15    nvec = [-e(:,2)./he, e(:,1)./he];
16    % assemble
17    FN = zeros(nel,ndof);
18    for p = 1:ng
19        pz = lambda(p,1)*z1 + lambda(p,2)*z2;
20        Dnu = sum(pde.Du(pz).*nvec,2);
21        FN = FN + weight(p)*Dnu*phi1(p,:);
22    end
23    FN = repmat(he,1,ndof).*FN;
24    ff = ff + accumarray(elem1(:), FN(:), [NNdof 1]);
25 end

```

这里, bdEdgeIdxN 为 Neumann 边的自然序号.

Dirichlet 边界条件如下处理.

```

1 %% Apply Dirichlet boundary conditions
2 bdNodeIdxD = bdStruct.bdNodeIdxD;
3 bdEdgeIdxD = bdStruct.bdEdgeIdxD;
4 id = [bdNodeIdxD; bdEdgeIdxD+N];
5 g_D = pde.g_D;
6 bdEdgeD = bdStruct.bdEdgeD;
7 isBdNode = false(NNdof,1); isBdNode(id) = true;
8 bdDof = (isBdNode); freeDof = (~isBdNode);
9 z1 = node(bdEdgeD(:,1),:); z2 = node(bdEdgeD(:,2),:);
10 zc = (z1+z2)/2;
11 nodeD = node(bdNodeIdxD,:);
12 wD = g_D(nodeD); wc = g_D(zc);
13 u = zeros(NNdof,1); u(bdDof) = [wD; wc];
14 ff = ff - kk*u;

```

我们修改了 getL2error.m 和 getH1error.m 函数, 它们可以求解 P1,P2,P3 元的相应误差. 函数文件和主程序略, 见 GitHub 上传文件 (PoissonP2.m 和 main_PoissonP2.m)

5.4 Poisson 方程的三阶有限元方法

P3-Lagrange 元的局部自由度有 10 个, 排列为:

$$\begin{cases} v(z_i), & i = 1, 2, 3; \\ v(a_i), & i = 1, 2, 3; \\ v(b_i), & i = 1, 2, 3; \\ v(z_c). \end{cases}$$

这里, a_i 第 i 条边的 $1/3$ 点, b_i 则是 $2/3$ 点, 而 z_c 是单元的重心. 整体自由度类似排列, 但要注意边必须事先规定好方向, 以保证局部单元上可确定 $1/3$ 点和 $2/3$ 点对应的整体编号.

稀疏装配指标如下

```

1 %% Get elem2dof
2 % auxstructure
3 auxT = auxstructure(node,elem);
4 edge = auxT.edge;
5 elem2edge = auxT.elem2edge;
6 % numbers
7 N = size(node,1); NT = size(elem,1); NE = size(edge,1);
8 Ndof = 10; NNdof = N + 2*NE + NT;
9 % sgnelem
10 v1 = [2 3 1]; v2 = [3 1 2];
11 bdEdgeIdx = bdStruct.bdEdgeIdx; E = false(NE,1); E(bdEdgeIdx) = 1;
12 sgnelem = sign(elem(:,v2)-elem(:,v1));
13 sgnbd = E(elem2edge); sgnelem(sgnbd) = 1;
14 sgnelem(sgnelem== -1) = 0;
15 elema = elem2edge + N*sgnelem + (N+NE)*(-sgnelem); % 1/3 point
16 elemb = elem2edge + (N+NE)*sgnelem + N*(-sgnelem); % 2/3 point
17 % local --> global
18 elem2dof = [elem, elema, elemb, (1:NT)' + N+2*NE];
19 ii = reshape(repmat(elem2dof, Ndof, 1), [], 1);
20 jj = repmat(elem2dof(:, ), Ndof, 1);

```

注意, 一维边必须确定好定向再编号, 这里 $sgnelem$ 起到该作用. 它是按单元给定的边符号, 若边是正定向 (人为规定的), 则对应 1, 否则对应 0. 为了方便处理边界, 边界边的定向始终规定为逆时针方向.

其他过程与 P2-元类似, 只需要修改基函数. 函数文件和主程序略, 见 GitHub 上传文件 (PoissonP3.m 和 main_PoissonP3.m)

后面的基于变分形式的程序中还会详细说明.

5.5 Poisson 方程的 Crouzeix-Raviart 非协调元方法

有限元问题与 (5.3) 类似, 只不过积分要按照单元求和理解.

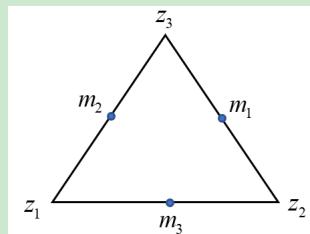


图 5.11. Crouzeix-Raviart 三角形元

CR 元在每个三角形上是一次多项式, 且在内部边的中点处连续. 它是最简单的非协调

元, 自由度为三边中点值. 易知中点 m_i 对应的节点基为

$$\phi_i = 1 - 2\lambda_i, \quad i = 1, 2, 3.$$

局部整体对应 `elem2dof` 显然就是辅助数据结构 `elem2edge`, 即按单元记录的边的序号. 注意到

$$\int_K \nabla \phi_j \cdot \nabla \phi_i dx dy = 4 \int_K \nabla \lambda_j \cdot \nabla \lambda_i dx dy,$$

它只是 P1-协调元情形的 4 倍. 刚度矩阵和载荷向量的计算与 Lagrange 元类似, 这里省略代码.

接下来, 我们处理 Neumann 边界条件. 设 $\Phi_j, j = 1, 2, \dots$ 是整体节点基, 则方程组右端的第 j 行要加上

$$v = \Phi_j : \int_{\Gamma_N} g_N v ds = \sum_{e \subset \Gamma_N} \int_e g_N v ds.$$

设 e 是 Neumann 边, 则它所在三角形的三个局部基函数都有贡献. 为了方便, 规定 e 为第一条边, 则有

$$\begin{cases} \phi_1|_e = (1 - 2\lambda_1)|_e = 1, \\ \phi_2|_e = (1 - 2\lambda_2)|_e = 1 - 2\lambda_{N,1}, \\ \phi_3|_e = (1 - 2\lambda_3)|_e = 1 - 2\lambda_{N,2}, \end{cases}$$

这里的 $\lambda_{N,1}$ 和 $\lambda_{N,2}$ 表示边 e 的起点和终点对应的线坐标函数. 把这三个“基函数”想象成对应边 e 上的三个点, 它们的局部整体对应是三角形边的整体序号 (按规定的顺序, 即视 e 为第一条边). 这样, Neumann 边界条件就可按照一维问题的方式进行装配.

注 5.6 若用中矩形公式近似 ϕ_2, ϕ_3 的项, 则积分为零. 而复合中矩形公式的误差为

$$\frac{1}{24} f''(\xi) h^2,$$

它与线性元的 L^2 误差阶是一致的. 正因为如此, 有时候直接忽略后两个基函数的贡献, 例如 iFEM 中就是如此.

我们事先在 `setboundary` 中给定 Neumann 边界的局部整体对应, 记为 `bdElem2edgeN`, 它与 `elem2edge` 类似, 只不过把 e 作为第一条边.

- `elem2edge` 行对应单元, 为此, 我们可先找到 Neumann 边所在的单元.

```

1 bdElemIdx = mod(i1(i1==i2),NT);
2 bdElemIdx(bdElemIdx==0) = NT;
3 bdElemIdxN = bdElemIdx(-IdxD);

```

- `totalEdge` 每 NT 行对应所有单元的一条边. `i1(i1==i2)` 给出的是边界边在 `totalEdge` 中的行号. 前两行给出的就是边界边的单元序号.
- `IdxD` 与 `bdEdge` 对应, 而 `bdElemIdx` 也与 `bdEdge` 对应, 故上面的过程给出了 Neumann 边所在的单元序号.
- 这样, 边界单元对应的 `elem2edge` 为

```

1 [~, ~, totalJ] = unique(totalEdge, 'rows');
2 elem2edge = reshape(totalJ, NT, 3);
3 bdElem2edgeN = elem2edge(bdElemIdxN, :);

```

- 接下来要调整边的局部序号, 以保证 e 是第一条边.

```

1 %idx1 = find((bdElem2edgeN(:,1)-bdEdgeIdxN)==0);
2 idx2 = ((bdElem2edgeN(:,2)-bdEdgeIdxN)==0);
3 idx3 = ((bdElem2edgeN(:,3)-bdEdgeIdxN)==0);
4 bdElem2edgeN(idx2,:) = bdElem2edgeN(idx2,[2,3,1]);
5 bdElem2edgeN(idx3,:) = bdElem2edgeN(idx3,[3,1,2]);

```

若 `bdElem2edgeN` 对应的第 1 条边是边界边, 则相应的序号由 `idx1` 给出. 此时的顺序符合规定, 不需要更改. 对其他两边, 根据局部顺序进行调整, 即最后两行.

根据上面的讨论, Neumann 边界条件如下装配.

```

1 %% Assemble Neumann boundary conditions
2 bdEdgeN = bdStruct.bdEdgeN;
3 if ~isempty(bdEdgeN)
4     % sparse assembling index
5     elem2dof1 = bdStruct.bdElem2edgeN;
6     % Gauss quadrature rule
7     [lambdaN, weightN] = quadpts1(3); ng = length(weightN);
8     % basis function
9     ndof = 3;
10    phi1(:,3) = 1-2*lambdaN(:,2)';
11    phi1(:,1) = ones(ng,1);
12    phi1(:,2) = 1-2*lambdaN(:,1)';
13    % nvec
14    z1 = node(bdEdgeN(:,1),:); z2 = node(bdEdgeN(:,2),:);
15    e = z1-z2;
16    nvec = [-e(:,2), e(:,1)];    % scaled
17    % assemble
18    FN = zeros(size(e,1),ndof);
19    for p = 1:ng
20        pz = lambdaN(p,1)*z1 + lambdaN(p,2)*z2;
21        Dnu = sum(pde.Du(pz).*nvec,2);
22        FN = FN + weightN(p)*Dnu*phi1(p,:);
23    end

```

```
24      ff = ff + accumarray(elem2dof1(:, FN(:, [NNdof 1]));
25 end
```

Dirichlet 边界条件比较容易处理, 此时固定的点是 Dirichlet 边的中点.

第六章 线弹性边值问题

本章考虑向量方程(也就是方程组), 即方程有多个未知函数, 经典的例子是线弹性边值问题.

6.1 线弹性边值问题简介

6.1.1 问题说明

设弹性体未受外力时所在区域为 $\Omega \subset \mathbb{R}^3$. 当它受体力 \mathbf{f} (在 Ω 中), 在 Ω 的一部分边界 Γ_1 受表面力 \mathbf{g} , 而在另一部分边界 Γ_0 上固定位移 $\mathbf{u} = \mathbf{0}$ 时, 弹性体就产生位移 \mathbf{u} . 在平衡状态下, 位移 \mathbf{u} 所满足的边值问题为

$$\begin{cases} -\partial_j \sigma_{ij}(\mathbf{u}) = f_i, & i = 1, 2, 3 \quad \text{in } \Omega, \\ \mathbf{u} = \mathbf{0} & \text{on } \Gamma_0, \\ \sigma_{ij}(\mathbf{u}) n_j = g_i, & i = 1, 2, 3 \quad \text{on } \Gamma_1, \end{cases} \quad (6.1)$$

这里约定: 凡在每一项中指标重复出现意味着从 1 到 3 (三维) 或 2 (二维) 求和. 上面的方程也可写为

$$\begin{cases} -\operatorname{div} \boldsymbol{\sigma} = \mathbf{f} & \text{in } \Omega, \\ \mathbf{u} = \mathbf{0} & \text{on } \Gamma_0, \\ \boldsymbol{\sigma} \mathbf{n} = \mathbf{g} & \text{on } \Gamma_1, \end{cases}$$

其中向量规定为列向量, 而 $\operatorname{div} \boldsymbol{\sigma}$ 是对矩阵 $\boldsymbol{\sigma}$ 的每行进行 (注意 $\boldsymbol{\sigma}$ 是对称矩阵).

在以上式子中, 第一式为平衡方程, 其中的 σ_{ij} 为应力张量, 它与应变张量 $\varepsilon_{ij}(\mathbf{u})$ 满足如下的本构关系 (均匀的各项同性弹性体的 Hooke 定律)

$$\sigma_{ij}(\mathbf{u}) = \sigma_{ji}(\mathbf{u}) = \lambda \varepsilon_{kk}(\mathbf{u}) \delta_{ij} + 2\mu \varepsilon_{ij}(\mathbf{u}), \quad (6.2)$$

$$\varepsilon_{ij}(\mathbf{u}) = \varepsilon_{ji}(\mathbf{u}) = \frac{1}{2} (\partial_j u_i + \partial_i u_j), \quad (6.3)$$

而 λ 和 μ 为 Lamé 系数. 注意, 这里 $\varepsilon_{kk}(\mathbf{u})$ 按规定是对指标求和的, 显然有 (标量)

$$\varepsilon_{kk}(\mathbf{u}) = \partial_k u_k = \operatorname{div} \mathbf{u}.$$

这样, 平衡方程也可写为如下的紧凑形式

$$-\operatorname{div} (\lambda(\operatorname{div} \mathbf{u}) \mathbf{I} + 2\mu \boldsymbol{\varepsilon}(\mathbf{u})) = \mathbf{f} \quad \text{in } \Omega.$$

把 (6.2)-(6.3) 代入 (6.1), 可获得平衡方程的另一形式

$$-\mu\Delta\mathbf{u} - (\lambda + \mu)\operatorname{grad}(\operatorname{div} \mathbf{u}) = \mathbf{f} \quad \text{in } \Omega, \quad (6.4)$$

有时候会直接考虑该方程, 因为其中每个算子都是熟悉的. 该形式常称为 Navier 形式, 因为它与 (Navier-) Stokes 方程有点像.

6.1.2 连续变分问题

设

$$V = \{\mathbf{v} \in H^1(\Omega)^3 : \mathbf{v} = \mathbf{0} \quad \text{on } \Gamma_0\}, \quad (6.5)$$

令 $\mathbf{v} = (v_1, v_2, v_3)^T \in V$, 在 (6.1) 的平衡方程的两边乘以 v_i , 有

$$\int_{\Omega} -\partial_j \sigma_{ij}(\mathbf{u}) v_i dx = \int_{\Omega} f_i v_i dx,$$

这里遵循求和约定, 即上式实际上求和. 分部积分有

$$-\int_{\partial\Omega} \sigma_{ij}(\mathbf{u}) v_i n_j ds + \int_{\Omega} \sigma_{ij}(\mathbf{u}) \partial_j v_i dx = \int_{\Omega} f_i v_i dx$$

或

$$-\int_{\partial\Omega} g_i v_i ds + \int_{\Omega} \sigma_{ij}(\mathbf{u}) \partial_j v_i dx = \int_{\Omega} f_i v_i dx.$$

由对称性,

$$\sigma_{ij}(\mathbf{u}) \partial_j v_i = \sigma_{ij}(\mathbf{u}) \frac{1}{2} (\partial_j v_i + \partial_i v_j) = \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}),$$

故

$$\int_{\Omega} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

变分形式可写为三种形式.

1. 第一种形式 (tensor1) 为: 求 $\mathbf{u} \in V$ 使得,

$$a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), \quad \mathbf{v} \in V, \quad (6.6)$$

式中,

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx, \quad \ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

文献中一般习惯引入如下记号

$$\mathbf{A} : \mathbf{B} = \sum_{ij} a_{ij} b_{ij}, \quad \mathbf{A} = (a_{ij}), \quad \mathbf{B} = (b_{ij}),$$

从而双线性形式可写为

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) dx.$$

2. 第二种形式. 注意到

$$\begin{aligned}
\sigma_{ij}(\mathbf{u})\varepsilon_{ij}(\mathbf{v}) &= (\lambda\varepsilon_{kk}(\mathbf{u})\delta_{ij} + 2\mu\varepsilon_{ij}(\mathbf{u}))\varepsilon_{ij}(\mathbf{v}) \\
&= (\lambda\partial_k u_k \delta_{ij} + 2\mu\varepsilon_{ij}(\mathbf{u}))\varepsilon_{ij}(\mathbf{v}) \\
&= 2\mu\varepsilon_{ij}(\mathbf{u})\varepsilon_{ij}(\mathbf{v}) + \lambda\partial_k u_k \varepsilon_{ii}(\mathbf{v}) \\
&= 2\mu\varepsilon_{ij}(\mathbf{u})\varepsilon_{ij}(\mathbf{v}) + \lambda\partial_k u_k \partial_i u_i,
\end{aligned}$$

双线性形式还可写为

$$\begin{aligned}
a(\mathbf{u}, \mathbf{v}) &= 2\mu \int_{\Omega} \varepsilon_{ij}(\mathbf{u})\varepsilon_{ij}(\mathbf{v}) dx + \lambda \int_{\Omega} \partial_i u_i \partial_j u_j dx, \\
&= 2\mu \int_{\Omega} \boldsymbol{\varepsilon}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) dx + \lambda \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx.
\end{aligned} \tag{6.7}$$

3. 第三种形式 (Navier). 也可采用 (6.4) 的形式, 具体写出来即

$$\begin{cases} -\mu\Delta u_1 - (\lambda + \mu)\partial_x(\operatorname{div} \mathbf{u}) = f_1, \\ -\mu\Delta u_2 - (\lambda + \mu)\partial_y(\operatorname{div} \mathbf{u}) = f_2, \end{cases}$$

注意 $\operatorname{div} \mathbf{u}$ 是标量. 第一式乘以 v_1 , 并分部积分有

$$\begin{aligned}
&\mu \left(\int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx - \int_{\partial\Omega} \partial_n u_1 v_1 ds \right) \\
&- (\lambda + \mu) \left(\int_{\partial\Omega} (\operatorname{div} \mathbf{u}) v_1 n_x ds - \int_{\Omega} (\operatorname{div} \mathbf{u}) \partial_x v_1 dx \right) = \int_{\Omega} f_1 v_1 dx.
\end{aligned}$$

类似可获得第二个式子对应的结果, 将它们相加有

$$\begin{aligned}
&\mu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx + (\lambda + \mu) \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx \\
&- \mu \int_{\partial\Omega} \partial_n \mathbf{u} \cdot \mathbf{v} ds - (\lambda + \mu) \int_{\partial\Omega} (\operatorname{div} \mathbf{u})(\mathbf{v} \cdot \mathbf{n}) ds = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx.
\end{aligned} \tag{6.8}$$

该形式仅考虑 Dirichlet 问题, 此时双线性形式为

$$a(\mathbf{u}, \mathbf{v}) = \mu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx + (\lambda + \mu) \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx.$$

iFEM 给出了该变分形式 Dirichlet 问题的程序 (elasticity.m).

注 6.1 前两种形式实际上是相同的, 本文统一为第二种形式, 并称其为 traction 形式.
第三种形式称为 Navier 形式.

注 6.2 第二种形式和第三种形式的双线性形式并不相等. 注意到

$$\boldsymbol{\varepsilon}(\mathbf{u}) = \nabla \mathbf{u} - \frac{1}{2}(\nabla \times \mathbf{u}) \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix},$$

我们有

$$\int_{\Omega} \boldsymbol{\varepsilon}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) dx = \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx - \frac{1}{2} \int_{\Omega} (\nabla \times \mathbf{u}) \cdot (\nabla \times \mathbf{v}) dx. \tag{6.9}$$

常记 $\operatorname{rot} \mathbf{u} = \nabla \times \mathbf{u}$.

6.1.3 有限元方法

以下考虑 $\Gamma_1 = \emptyset$ 的情形. 协调一次元空间为

$$V_h = \left\{ \mathbf{v} \in (H_0^1(\Omega))^2 : \mathbf{v}|_K \in (\mathbb{P}_1(K))^2, K \in \mathcal{T}_h \right\},$$

相应的有限元问题为: 求 $\mathbf{u}_h \in V_h$ 使得

$$a(\mathbf{u}_h, \mathbf{v}) = \ell(\mathbf{v}), \quad \mathbf{v} \in V_h.$$

定理 6.1 设 \mathbf{u} 和 \mathbf{u}_h 分别为连续变分问题和近似变分问题的解, 则

$$\|\mathbf{u} - \mathbf{u}_h\|_{1,\Omega} \lesssim (2\mu + \lambda)h|\mathbf{u}|_{2,\Omega}.$$

注 6.3 对几乎不可压缩的材料, $\lambda \gg \mu$. 从上面的误差估计以及数值实例可以看到, 当 $\lambda \rightarrow \infty$ 时, 协调一次元方法不再收敛, 称为闭锁 (locking) 现象. 事实上, 我们的确可以构造出不收敛的例子.

6.2 刚度矩阵与载荷向量的装配

对向量方程, 因含有两个未知函数, 装配的过程与它们的排列顺序相关. 为了方便, 考虑如下变分形式

$$a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}),$$

式中,

$$a(\mathbf{u}, \mathbf{v}) = \sum_{i,j=1}^2 \int_{\Omega} u_i v_j dx, \quad \ell(\mathbf{v}) = \sum_{i=1}^2 \int_{\Omega} f_i v_i dx.$$

6.2.1 单元的向量法分析

对 $\mathbf{u} = [u_1, u_2]^T$, 为了避免下标的混乱, 我们记 $\bar{u} = u_1$ 和 $\underline{u} = u_2$, 相应的节点基展开为

$$\bar{u} = \bar{u}_1 \varphi_1 + \cdots + \bar{u}_n \varphi_n = \Psi \bar{U}, \quad \underline{u} = \underline{u}_1 \varphi_1 + \cdots + \underline{u}_n \varphi_n = \Psi \underline{U},$$

其中,

$$\Psi = [\varphi_1, \dots, \varphi_n]^T, \quad \bar{U} = [\bar{u}_1, \dots, \bar{u}_n]^T, \quad \underline{U} = [\underline{u}_1, \dots, \underline{u}_n]^T.$$

于是,

$$\mathbf{u} = \sum_{i=1}^n \bar{u}_i \bar{\varphi}_i + \sum_{i=1}^n \underline{u}_i \underline{\varphi}_i, \tag{6.10}$$

式中,

$$\bar{\varphi}_j = \begin{bmatrix} \varphi_j \\ 0 \end{bmatrix}, \quad \underline{\varphi}_j = \begin{bmatrix} 0 \\ \varphi_j \end{bmatrix}, \quad j = 1, \dots, n.$$

令

$$N_j = [\bar{\varphi}_j, \underline{\varphi}_j] = \begin{bmatrix} \bar{\varphi}_j^T \\ \underline{\varphi}_j^T \end{bmatrix}, \quad j = 1, \dots, n,$$

它是对称的, 对向量法编程, 通常按行考虑 (符合矩阵向量运算). 显然有

$$\mathbf{u} = \sum_{i=1}^n N_i U_i, \quad U_i = \begin{bmatrix} \bar{u}_i \\ \underline{u}_i \end{bmatrix}$$

或

$$\mathbf{u} = \begin{bmatrix} \bar{u} \\ \underline{u} \end{bmatrix} = N U,$$

式中,

$$N = [N_1, \dots, N_n], \quad U = [U_1; \dots; U_n],$$

这里向量之间的分号表示按列拉直, 与 MATLAB 的运算一致.

这样, 双线性形式和右端可写为

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \mathbf{v}^T \mathbf{u} dx = V^T \int_{\Omega} N^T N dx U = V^T A U, \quad A = \int_{\Omega} N^T N dx, \\ \ell(\mathbf{v}) &= \int_{\Omega} \mathbf{v}^T \mathbf{f} dx = V^T \int_{\Omega} N^T \mathbf{f} dx = V^T F, \quad F = \int_{\Omega} N^T \mathbf{f} dx, \end{aligned}$$

于是

$$A U = F.$$

可以看到, 在把向量 $U_i = \begin{bmatrix} \bar{u}_i \\ \underline{u}_i \end{bmatrix}$ 视为变量的情况下, 分析过程与标量情形完全相同.

事实上, 对三角形单元 β , 设三个顶点分别为 $\delta_1, \delta_2, \delta_3$, 取 φ_i 为局部基函数, 同上有

$$A_{\beta} U_{\beta} = F_{\beta},$$

式中,

$$\begin{aligned} A_{\beta} &= \int_{\beta} N^T N dx, \quad F_{\beta} = \int_{\beta} N^T \mathbf{f} dx, \\ N &= [N_1, N_2, N_3], \quad U_{\beta} = [U_1; U_2; U_3]. \end{aligned}$$

单元矩阵和载荷向量可分块为

$$\begin{aligned} A_{\beta} &= (K_{st})_{3 \times 3}, \quad K_{ij} = \int_{\beta} N_s^T N_t dx \in \mathbb{R}^{2 \times 2}, \\ F_{\beta} &= (F_s)_{3 \times 1}, \quad F_s = \int_{\beta} N_s^T \mathbf{f} dx \in \mathbb{R}^2. \end{aligned}$$

此时, K_{ij} 和 F_i 相当于标量情形的 k_{ij} 和 f_i .

6.2.2 sparse 装配指标

把单元刚度矩阵按分量记为 $A_\beta = (k_{ij})_{6 \times 6}$, 可以给出相应的 `sparse` 装配指标. 即首先给出 k_{11} 所有单元的 (i, j, s) , 并记为 i_{11}, j_{11}, s_{11} . 接着按单元矩阵的行给出其他分量的, 排列为

$$\begin{bmatrix} i_{11} & j_{11} & s_{11} \\ i_{12} & j_{12} & s_{12} \\ \vdots & \vdots & \vdots \\ i_{66} & j_{66} & s_{66} \end{bmatrix}.$$

相较于按分量进行分块的方式, 上面的做法不利于 MATLAB 的向量运算 (或者说为了用向量运算, 一些数据需要稍复杂的存储).

若把变量按标量编程法排列, 即

$$a_\beta(\mathbf{u}, \mathbf{v}) \rightarrow \begin{bmatrix} A_{11}^\beta & A_{12}^\beta \\ A_{21}^\beta & A_{22}^\beta \end{bmatrix} \begin{bmatrix} \bar{U}_\beta \\ U_\beta \end{bmatrix},$$

则容易发现

$$A_{ij}^\beta = (K_{st}(i, j))_{3 \times 3}, \quad 1 \leq i, j \leq 2,$$

即把每个 K_{st} 的 (i, j) 位置元素拿出来组成的矩阵恰是分块矩阵的 A_{ij}^β . 本文称这种分块的方式为标量法编程.

1. 单元分析也可按分量进行分析, 但用向量分析法更加清楚.
2. 分块单元刚度矩阵的向量稀疏指标也容易获得. 若直接用标量法编程, 则有

$$\bar{U}_\beta \leftrightarrow \text{elem}(:), \quad U_\beta \leftrightarrow \text{elem}(:) + N.$$

显然向量法只需要改为

$$\bar{U}_\beta \leftrightarrow 2 * \text{elem}(:) - 1, \quad U_\beta \leftrightarrow 2 * \text{elem}(:).$$

本文只采用前者.

刚度矩阵

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

的每个块都可直接按照标量情形的方法进行装配, 即用 `sparse(ii, jj, ss, N, N)` 生成, 其中的 `ii, jj` 是通用的. 而 `ss` 如下获得: 将单元刚度矩阵行拉直, 逐个单元拼成一个矩阵 (每行对应一个单元的拉直向量), 然后再拉直为一个列向量. 若把这些块给出的拉直向量分别记为 `ss11, ss12, ss21, ss22`, 则可如下装配

```

1 A11 = sparse(ii,jj,ss11,N,N); A12 = sparse(ii,jj,ss12,N,N);
2 A21 = sparse(ii,jj,ss21,N,N); A22 = sparse(ii,jj,ss22,N,N);
3 A = [A11,A12; A21,A22];

```

为了避免对稀疏矩阵进行运算, 上面分块装配可如下改进

CODE 6.1. 向量方程的 `sparse` 装配指标

```

1 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
2 ii21 = ii+N; jj21 = jj; ii22 = ii+N; jj22 = jj+N;
3 ii = [ii11; ii12; ii21; ii22];
4 jj = [jj11; jj12; jj21; jj22];
5 ss = [ss11; ss12; ss21; ss22];
6 A = sparse(ii,jj,ss,2*N,2*N);

```

单元载荷 F_β 分成三块, 每块 F_s 都是两行的向量. 类似刚度矩阵,

$$F_i^\beta = (F_s(i))_{3 \times 1}, \quad i = 1, 2,$$

即把每个 F_s 的第 i 个元素拿出来组成的向量恰是分块情形的 F_i^β . 给定每块的向量 $F1, F2$, 则如下装配

```

1 ff = accumarray([elem(:); elem(:)+N], [F1(:); F2(:)], [2*N 1]);

```

6.2.3 双线性分量的配对

我们用系统方程分析法来考察一下分量配对 (v_i, u_j) 对应的单元矩阵和单元向量, 从而能够明确装配的具体对应. 对 \mathbf{u} 进行分块形式的展开 (即式 (6.10)), 变分形式对应如下若干个方程

$$\begin{cases} a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), & \mathbf{v} = \bar{\varphi}_j, \quad j = 1, \dots, N; \\ a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), & \mathbf{v} = \underline{\varphi}_j, \quad j = 1, \dots, N. \end{cases}$$

注意到 $\bar{\varphi}_j = [\varphi_j, 0]^T$, 第一行方程只会留下检验函数 $\bar{v} = \varphi_j$ 对应的项, 从而

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \bar{u} \bar{v} dx + \int_{\Omega} \underline{u} \bar{v} dx, \quad \bar{v} = \varphi_j,$$

即

$$a(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^N \int_{\Omega} \varphi_j \varphi_i dx \bar{u}_i + \sum_{i=1}^N \int_{\Omega} \varphi_j \varphi_i dx \underline{u}_i,$$

从而第一行双线性形式对应

$$a(\mathbf{u}, \mathbf{v}), \quad \mathbf{v} = \bar{\varphi}_j, \quad j = 1, \dots, N \quad \rightarrow \quad A_{11} \bar{U} + A_{12} \underline{U} = [A_{11}, A_{12}] \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix},$$

这里 $A_{11} = (\varphi_j, \varphi_i)$ 对应 (\bar{v}, \bar{u}) , 而 $A_{12} = (\varphi_j, \varphi_i)$ 对应 (\bar{v}, u) . 类似地, 第二行方程对应

$$a(\mathbf{u}, \mathbf{v}), \quad \mathbf{v} = \underline{\varphi}_j, \quad j = 1, \dots, N \quad \rightarrow \quad A_{21}\bar{U} + A_{22}\underline{U} = [A_{21}, A_{22}] \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix},$$

这里 A_{21} 对应 (\underline{v}, \bar{u}) , 而 A_{22} 对应 $(\underline{v}, \underline{u})$. 两行方程组拼接起来就是

$$a(\mathbf{u}, \mathbf{v}) \quad \rightarrow \quad \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}.$$

从上面的分析可以看到,

$$\begin{aligned} (v_1, u_1) \quad &\rightarrow \quad \begin{bmatrix} A_{11} & O \\ O & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}; \quad (v_1, u_2) \quad \rightarrow \quad \begin{bmatrix} O & A_{12} \\ O & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}; \\ (v_2, u_1) \quad &\rightarrow \quad \begin{bmatrix} O & O \\ A_{21} & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}; \quad (v_2, u_2) \quad \rightarrow \quad \begin{bmatrix} O & O \\ O & A_{22} \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}. \end{aligned}$$

而 (v_i, u_j) 对应的 A_{ij} 恰是单个函数情形给出的刚度矩阵. 为此, 刚度矩阵的装配算法如下

算法 1 向量方程刚度矩阵的装配

1. 设 $\mathbf{u} = [u_1, \dots, u_d]^T$, $\mathbf{v} = [v_1, \dots, v_d]^T$ (这里 $d = 2$), 而 U_i 是 u_i 在节点基下的展开向量, 则刚度矩阵形如

$$\begin{bmatrix} A_{11} & \cdots & A_{1d} \\ \vdots & \ddots & \vdots \\ A_{d1} & \cdots & A_{dd} \end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix} U_1 \\ \vdots \\ U_d \end{bmatrix}.$$

2. A_{ij} 是对应 (v_i, u_j) 的双线性形式给出的刚度矩阵 (标量情形), 按前面介绍的装配算法实施.
-

类似给出载荷向量的装配, 右端为

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx = \sum_{i=1}^d \int_{\Omega} f_i v_i dx.$$

算法 2 向量方程载荷向量的装配

1. 设 $\mathbf{f} = [f_1, \dots, f_d]^T$, 而 U_i 是 u_i 在节点基下的展开向量, 则载荷向量形如

$$\begin{bmatrix} F_1 \\ \vdots \\ F_d \end{bmatrix} \leftrightarrow \begin{bmatrix} U_1 \\ \vdots \\ U_d \end{bmatrix}.$$

2. F_i 是对应 (v_i, f_i) 的线性形式给出的载荷向量 (标量情形), 按前面介绍的装配算法实施.

6.3 Traction 形式的变分问题

6.3.1 刚度矩阵的计算

单元变分形式为

$$\begin{aligned} a_K(\mathbf{u}, \mathbf{v}) &= 2\mu \int_K \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx + \lambda \int_K (\partial_i u_i)(\partial_j v_j) dx \\ &=: 2\mu I + \lambda J, \end{aligned}$$

注意求和约定.

第一部分 (v_i, u_j) 的配对如下

$$\left\{ \begin{array}{l} (v_1, u_1) : \int_K (\partial_1 v_1 \partial_1 u_1 + 0.5 \cdot \partial_2 v_1 \partial_2 u_1) dx, \\ (v_1, u_2) : \int_K 0.5 \cdot \partial_2 v_1 \partial_1 u_2 dx, \\ (v_2, u_1) : \int_K 0.5 \cdot \partial_1 v_2 \partial_2 u_1 dx, \\ (v_2, u_2) : \int_K (\partial_2 v_2 \partial_2 u_2 + 0.5 \cdot \partial_1 v_2 \partial_1 u_2) dx. \end{array} \right.$$

为了方便, 我们先求出所有基函数不同导数的配对结果.

- 注意到 $\partial_j \phi_s$ 是常数, 我们有

$$(\partial_i \phi_s, \partial_j \phi_t)_\beta = \partial_i \phi_s \cdot \partial_j \phi_t \cdot |\beta|,$$

从而要计算局部节点基, 也就是面积坐标函数的导数, 以及单元的面积, 这些计算在前面已经说明过, 例如节 12.2.4. 计算中用三维数组 `dphi` 存储三个基函数的导数, 如下

```

1 function [Dphi,area] = gradbasis(node,elem)
2
3 z1 = node(elem(:,1),:);
4 z2 = node(elem(:,2),:);
5 z3 = node(elem(:,3),:);
6 e1 = z2-z3; e2 = z3-z1; e3 = z1-z2;
7 area = 0.5*(-e3(:,1).*e2(:,2)+e3(:,2).*e2(:,1));
8
9 grad1 = [e1(:,2), -e1(:,1)]./(2*area); % stored in rows
10 grad2 = [e2(:,2), -e2(:,1)]./(2*area);
11 grad3 = -(grad1+grad2);
12
13 NT = size(elem,1);
14 Dphi(1:NT,:,1) = grad1;
15 Dphi(1:NT,:,2) = grad2;
16 Dphi(1:NT,:,3) = grad3;

```

这里, grad1 的行对应单元.

- 所有单元的 $(\partial_i \phi_s, \partial_j \phi_t)_\beta$ 用元胞数组 Dbase 存储, 它是 2×2 的, 每块存储一个导数配对. 根据装配的说明, 单元矩阵按行拉直, 然后逐个单元拼在一起. 为此这里每块也遵循这个规定. 程序如下

```

1 Dbase = cell(2,2);
2 for i = 1:2
3     for j = 1:2
4         k11 = Dphi(:,i,1).*Dphi(:,j,1).*area;
5         k12 = Dphi(:,i,1).*Dphi(:,j,2).*area;
6         k13 = Dphi(:,i,1).*Dphi(:,j,3).*area;
7         k21 = Dphi(:,i,2).*Dphi(:,j,1).*area;
8         k22 = Dphi(:,i,2).*Dphi(:,j,2).*area;
9         k23 = Dphi(:,i,2).*Dphi(:,j,3).*area;
10        k31 = Dphi(:,i,3).*Dphi(:,j,1).*area;
11        k32 = Dphi(:,i,3).*Dphi(:,j,2).*area;
12        k33 = Dphi(:,i,3).*Dphi(:,j,3).*area;
13        K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in rows
14        Dbase{i,j} = K(:); % straighten
15    end
16 end

```

根据分块的讨论, 有

```

1 % ----- Stiffness matrix for (Eij(u):Eij(v)) -----
2 ss11 = Dbase{1,1}+0.5*Dbase{2,2}; ss12 = 0.5*Dbase{2,1};
3 ss21 = 0.5*Dbase{1,2}; ss22 = Dbase{2,2}+0.5*Dbase{1,1};
4 A11 = sparse(ii,jj,ss11,N,N); A12 = sparse(ii,jj,ss12,N,N);
5 A21 = sparse(ii,jj,ss21,N,N); A22 = sparse(ii,jj,ss22,N,N);

```

```

6 A = [A11,A12; A21,A22];
7 A = 2*mu*A;

```

根据装配的说明, 上面的分块写法也可直接用 `sparse` 实现, 如下

```

1 ii11 = ii;    jj11 = jj;    ii12 = ii;    jj12 = jj+N;
2 ii21 = ii+N; jj21 = jj;    ii22 = ii+N; jj22 = jj+N;
3
4 ss11 = Dbase{1,1}+0.5*Dbase{2,2};  ss12 = 0.5*Dbase{2,1};
5 ss21 = 0.5*Dbase{1,2};           ss22 = Dbase{2,2}+0.5*Dbase{1,1};
6 ii = [ii11; ii12; ii21; ii22];
7 jj = [jj11; jj12; jj21; jj22];
8 ss = [ss11; ss12; ss21; ss22];
9 A = sparse(ii,jj,ss,2*N,2*N);
10 A = 2*mu*A;

```

现在考虑第二部分, 即

$$\lambda J = \lambda \int_K (\partial_i u_i)(\partial_j v_j) dx,$$

注意指标的求和约定. 明确写出来为

$$J = \int_K (\partial_1 v_1 \partial_1 u_1 + \partial_2 v_2 \partial_1 u_1 + \partial_1 v_1 \partial_2 u_2 + \partial_2 v_2 \partial_2 u_2) dx.$$

它的装配可根据前面的分量配对给出.

```

1 % ----- Stiffness matrix for (div u,div v) -----
2 ss11 = Dbase{1,1};           ss12 = Dbase{1,2};
3 ss21 = Dbase{2,1};           ss22 = Dbase{2,2};
4 B11 = sparse(ii,jj,ss11,N,N); B12 = sparse(ii,jj,ss12,N,N);
5 B21 = sparse(ii,jj,ss21,N,N); B22 = sparse(ii,jj,ss22,N,N);
6 B = [B11,B12; B21,B22];
7 B = lambda*B;

```

或直接装配

```

1 % (div u,div v)
2 ss11 = Dbase{1,1};           ss12 = Dbase{1,2};
3 ss21 = Dbase{2,1};           ss22 = Dbase{2,2};
4 ss = [ss11; ss12; ss21; ss22];
5 B = sparse(ii,jj,ss,2*N,2*N);
6 B = lambda*B;

```

6.3.2 载荷向量的计算

右端为

$$\int_{\beta} \mathbf{f} \cdot \mathbf{v} dx = \int_{\beta} \mathbf{v}^T \mathbf{f} dx = V_{\beta}^T \int_{\beta} N^T \mathbf{f} dx = V_{\beta}^T F_{\beta},$$

单元载荷向量为

$$F_\beta = \int_{\beta} N^T \mathbf{f} dx = (F_i)_{3 \times 1},$$

式中,

$$F_i = \int_{\beta} N_i^T \mathbf{f} dx = \int_{\beta} \begin{bmatrix} \phi_i & 0 \\ 0 & \phi_i \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} dx = \int_{\beta} \begin{bmatrix} f_1 \phi_i \\ f_2 \phi_i \end{bmatrix} dx.$$

我们先计算第一个分量所有单元的结果, 再计算第二个分量的, 即分别考虑

$$f_i \phi_1, f_i \phi_2, f_i \phi_3, i = 1, 2,$$

这对应的标量情形的载荷向量, 与常规计算一致.

用前面介绍的三角形上的 Gauss 求积公式计算, 注意此时的被积函数为(单元排成一行)

$$f_i \phi_1, f_i \phi_2, f_i \phi_3, i = 1, 2.$$

对 $\mathbf{f} = [f_1, f_2]$, 可如下计算

```

1 % Gauss quadrature rule
2 [lambda,weight] = quadpts(2);
3 F1 = zeros(NT,3); F2 = zeros(NT,3);
4 for p = 1:length(weight)
5     pxy = lambda(p,1)*node(elem(:,1),:) ...
6         + lambda(p,2)*node(elem(:,2),:) ...
7         + lambda(p,3)*node(elem(:,3),:);
8     fxy = f(pxy); % fxy = [f1xy,f2xy]
9     F1 = F1 + weight(p)*repmat(fxy(:,1),1,3).*lambda(p,:);
10    F2 = F2 + weight(p)*repmat(fxy(:,2),1,3).*lambda(p,:);
11 end
12 F1 = repmat(area,1,3).*F1; % F = area.*F;
13 F2 = repmat(area,1,3).*F2;
14 ff = accumarray([elem(:); elem(:)+N], [F1(:); F2(:)], [2*N 1]);

```

6.3.3 边界条件的处理

边界定义函数见前面介绍的 setboundary.m.

在 Γ_1 上有附加载荷, 所得边界条件为

$$\sigma \mathbf{n} = \mathbf{g} \quad \text{on } \Gamma_1,$$

不妨称为 Neumann 边界条件. 变分形式的右端要加上

$$\int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

设 $\gamma = (\delta_1, \delta_2)$ 是边界边, 相应的局部节点基为 $\tilde{\phi}_1, \tilde{\phi}_2$. 记

$$U_\gamma = [\bar{u}_1, \underline{u}_1, \bar{u}_2, \underline{u}_2],$$

$$\tilde{N} = [\tilde{N}_1, \tilde{N}_2], \quad \tilde{N}_i = \tilde{\phi}_i \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

则

$$\int_\gamma \mathbf{g} \cdot \mathbf{v} ds = \int_\gamma \mathbf{v}^T \mathbf{g} ds = V_\gamma^T \int_\gamma \tilde{N}^T \mathbf{g} ds =: V_\gamma^T F_\gamma,$$

式中,

$$F_\gamma = \int_\gamma \tilde{N}^T \mathbf{g} ds = [F_1, F_2], \quad F_i = \int_\gamma \begin{bmatrix} g_1 \tilde{\phi}_i \\ g_2 \tilde{\phi}_i \end{bmatrix} ds.$$

由梯形公式, 有

$$F_i = \int_\gamma \mathbf{g} \tilde{\phi}_i ds = \int_\gamma \boldsymbol{\sigma} \mathbf{n} \tilde{\phi}_i ds = \frac{h_\gamma}{2} \left((\boldsymbol{\sigma} \mathbf{n} \tilde{\phi}_i)(\delta_1) + (\boldsymbol{\sigma} \mathbf{n} \tilde{\phi}_i)(\delta_2) \right) = \frac{h_\gamma}{2} (\boldsymbol{\sigma} \mathbf{n})(\delta_i),$$

其中

$$h_\gamma = |\delta_2 - \delta_1| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

注意到

$$h_\gamma \boldsymbol{\sigma} \mathbf{n} = \boldsymbol{\sigma}(h_\gamma \mathbf{n}),$$

而 $h_\gamma \mathbf{n}$ 恰是定向边 γ 顺时针旋转 90° 或 $-\gamma$ 逆时针旋转 90° 所得. 这样, 设 $-\gamma$ 的坐标为 (a, b) , 则 $h_\gamma \mathbf{n}$ 的坐标为 $(-b, a)$.

类似载荷向量的装配, 先计算所有 F_i 的第一个分量并放在一起, 再计算第二个分量的放在一起. 我们在 elasticitydata.m 中定义

$$g_N = [\sigma_{11}(x, y), \sigma_{22}(x, y), \sigma_{12}(x, y)],$$

结合一维问题的装配算法, Neumann 边界条件可如下编程

```

1 %% Assemble Neumann boundary conditions
2 bdEdgeN = bdStruct.bdEdgeN;
3 if ~isempty(bdEdgeN)
4     g_N = pde.g_N;
5     z1 = node(bdEdgeN(:,1),:); z2 = node(bdEdgeN(:,2),:);
6     e = z1-z2; % e = z2-z1
7     ne = [-e(:,2),e(:,1)]; % scaled ne
8     Sig1 = g_N(z1); Sig2 = g_N(z2);
9     F11 = sum(ne.*Sig1(:,[1,3]),2)./2; F12 = sum(ne.*Sig2(:,[1,3]),2)./2; % g1
10    F21 = sum(ne.*Sig1(:,[3,2]),2)./2; F22 = sum(ne.*Sig2(:,[3,2]),2)./2;
11    FN = [F11,F12,F21,F22];
12    ff = ff + accumarray([bdEdgeN(:); bdEdgeN(:)+N], FN(:), [2*N 1]);
13 end

```

Dirichlet 边界条件类似标量情形处理, 如下

```
1 %% Apply Dirichlet boundary conditions
2 g_D = pde.g_D; bdNodeIdx = bdStruct.bdNodeIdx;
3 id = [bdNodeIdx; bdNodeIdx+N];
4 isBdNode = false(2*N,1); isBdNode(id) = true;
5 bdDof = (isBdNode); freeDof = (~isBdNode);
6 pD = node(bdNodeIdx,:);
7 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
8 ff = ff - kk*u;
```

6.3.4 程序整理

设左右边界是 Neumann 边界条件, 主程序如下

CODE 6.2. main_elasticity.m

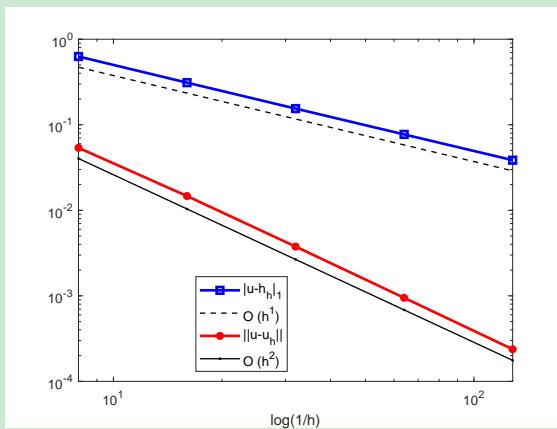
```
1 clc; clear; close all;
2 %% Parameters
3 maxIt = 5;
4 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
5 ErrL2 = zeros(maxIt,1);
6 ErrH1 = zeros(maxIt,1);
7
8 %% Generate an initial mesh
9 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
10 Nx = 4; Ny = 4; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
11 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
12 bdNeumann = 'abs(y-0)<1e-4 | abs(x-1)<1e-4'; % string for Neumann
13
14 %% Get the PDE data
15 lambda = 1; mu = 1;
16 para.lambda = lambda; para.mu = mu;
17 pde = elasticitydata(para);
18
19 %% Finite element method
20 for k = 1:maxIt
21     % refine mesh
22     [node,elem] = uniformrefine(node,elem);
23     % set boundary
24     bdStruct = setboundary(node,elem,bdNeumann);
25     % set up solver type
26     option.solver = 'mg';
27     option.J = k+1;
28     % solve the equation
29     uh = elasticity(node,elem,pde,bdStruct,option);
30     uh = reshape(uh,[],2);
31     % record and plot
32     NNdof(k) = length(uh);
```

```

33     h(k) = 1/(sqrt(size(node,1))-1);
34
35     if NNdof(k)<2e3
36         figure(1);
37         showresult(node,elem,pde.uexact,uh(:,1));
38         pause(1);
39     end
40
41     % compute error
42     tru = eye(2); trDu = eye(4);
43     errL2 = zeros(1,2); errH1 = zeros(1,2); % square
44     for id = 1:2
45         uid = uh(:,id);
46         u = @(pz) pde.uexact(pz)*tru(:, id);
47         Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2:id);
48         errL2(id) = getL2error(node, elem, u, uid);
49         errH1(id) = getH1error(node, elem, Du, uid);
50     end
51
52
53 %% Plot convergence rates and display error table
54 figure(2);
55 showrateh(h,ErrH1,ErrL2);
56
57 fprintf('\n');
58 disp('Table: Error')
59 colname = {'#Dof','h','||u-u_h||','|u-u_h|_1'};
60 disptable(colname,NNdof,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
61
62 %% Conclusion
63 %
64 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
65 % (2nd order) is observed.

```

注意, 程序中通过截取向量的分量来获得分量的误差 (回忆初等变换的特点).



6.4 Navier 形式的变分问题

变分形式 (6.8) 的程序只需要对前面的进行简单修改即可. 需要注意的是, 此种形式产生的边界项为

$$\mu \int_{\partial\Omega} \partial_n \mathbf{u} \cdot \mathbf{v} ds + (\lambda + \mu) \int_{\partial\Omega} (\operatorname{div} \mathbf{u})(\mathbf{v} \cdot \mathbf{n}) ds,$$

给出相应的边界条件我们也可容易地处理边界项. 但弹性问题中通常不附加上面类型的边界条件, 为此, 第三种形式只考虑 Dirichlet 边界条件, 此时检验函数在边界上为零.

主程序为 (数据文件为 elasticitydata1.m, L 型区域, 用 MATLAB 的 PDE 工具箱生成)

CODE 6.3. main_elasticityNavier.m

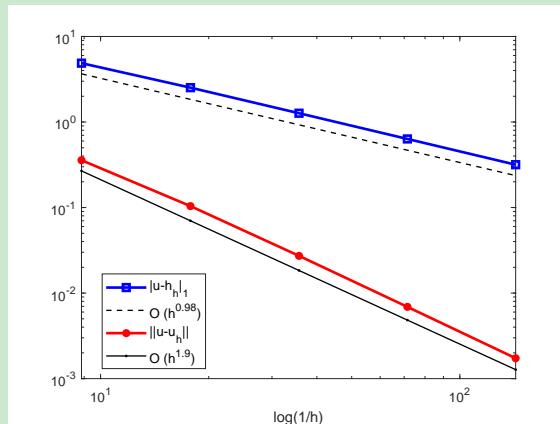
```
1 clc; clear; close all;
2 %% Parameters
3 maxIt = 5;
4 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
5 ErrL2 = zeros(maxIt,1);
6 ErrH1 = zeros(maxIt,1);
7
8 %% Generate an initial mesh
9 g = [2 2 2 2 2 2    % decomposed geometry matrix
10      0 1 1 -1 -1 0
11      1 1 -1 -1 0 0
12      0 0 1 1 -1 -1
13      0 1 1 -1 -1 0
14      1 1 1 1 1 1
15      0 0 0 0 0 0];
16 [p,e,t] = initmesh(g,'hmax',0.5); % initial mesh
17 bdNeumann = []; % only Dirichlet condition for elasticity_Navier
18
19 %% Get the PDE data
20 lambda = 1; mu = 1;
21 para.lambda = lambda; para.mu = mu;
22 pde = elasticitydata1(para);
23
24 %% Finite element method
25 for k = 1:maxIt
26     % refine mesh
27     [p,e,t] = refinemesh(g,p,e,t);
28     node = p'; elem = t(1:3,:)';
29     % set boundary
30     bdStruct = setboundary(node,elem,bdNeumann);
31     % solve the equation
32     uh = elasticityNavier(node,elem,pde,bdStruct);
33     uh = reshape(uh,[],2);
34     % record and plot
35     NNdof(k) = length(uh);
36     h(k) = 1/(sqrt(size(node,1))-1);
```

```

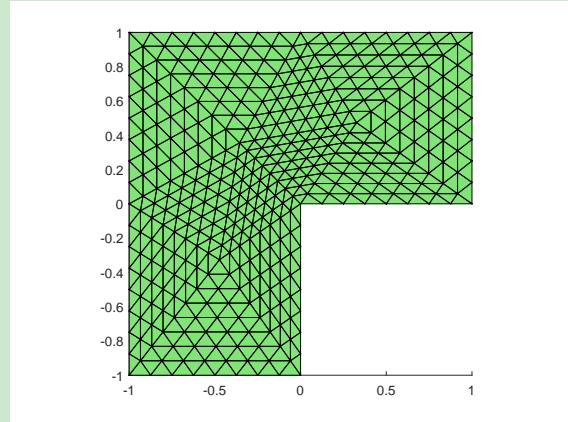
37 if NNdof(k)<2e3
38     figure(1);
39     showresult(node,elem,pde.uexact,uh(:,1));
40     pause(1);
41 end
42 % compute error
43 tru = eye(2); trDu = eye(4);
44 errL2 = zeros(1,2); errH1 = zeros(1,2); % square
45 for id = 1:2
46     uid = uh(:,id);
47     u = @(pz) pde.uexact(pz)*tru(:, id);
48     Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2:id);
49     errL2(id) = getL2error(node, elem, u, uid);
50     errH1(id) = getH1error(node, elem, Du, uid);
51 end
52 ErrL2(k) = norm(errL2);
53 ErrH1(k) = norm(errH1);
54 end
55
56 %% Plot convergence rates and display error table
57 figure(2);
58 showrateh(h,ErrH1,ErrL2);
59
60 fprintf('\n');
61 disp('Table: Error')
62 colname = {'#Dof','h','||u-u_h||','||u-u_h||_1'};
63 disptable(colname,NNdof,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
64
65 %% Conclusion
66 %
67 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
68 % (2nd order) is observed.

```

结果为



网格剖分如下



变分形式 (6.8) 的程序只需要对前面的进行简单修改即可, 如下

CODE 6.4. elasticityNavier.m

```

1 function u = elasticity_Navier(node, elem, pde, bdStruct)
2 %elasticity_Navier solves linear elasticity equation of Navier form using P1 element
3 %
4 %      u = [u1, u2]
5 %      -mu \Delta u - (lambda + mu)*grad(div(u)) = f in \Omega
6 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D.
7 %
8 % Copyright (C) Terence Yu.
9
10 N = size(node,1); NT = size(elem,1); Ndof = 3;
11 mu = pde.mu; lambda = pde.lambda; f = pde.f;
12
13 %% Compute (Dbase,Djbase)
14 [Dphi,area] = gradbasis(node,elem);
15 Dbase = cell(2,2);
16 for i = 1:2
17     for j = 1:2
18         k11 = Dphi(:,i,1).*Dphi(:,j,1).*area;
19         k12 = Dphi(:,i,1).*Dphi(:,j,2).*area;
20         k13 = Dphi(:,i,1).*Dphi(:,j,3).*area;
21         k21 = Dphi(:,i,2).*Dphi(:,j,1).*area;
22         k22 = Dphi(:,i,2).*Dphi(:,j,2).*area;
23         k23 = Dphi(:,i,2).*Dphi(:,j,3).*area;
24         k31 = Dphi(:,i,3).*Dphi(:,j,1).*area;
25         k32 = Dphi(:,i,3).*Dphi(:,j,2).*area;
26         k33 = Dphi(:,i,3).*Dphi(:,j,3).*area;
27         K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in rows
28         Dbase{i,j} = K(:); % straighten
29     end
30 end
31
32 %% Get sparse assembling index
33 ii = reshape(repmat(elem, Ndof,1), [], 1);

```

```

34 jj = repmat(elem(:, ), Ndof, 1);
35 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
36 ii21 = ii+N; jj21 = jj; ii22 = ii+N; jj22 = jj+N;
37
38 %% Assemble stiffness matrix
39 % (grad u,grad v)
40 ss11 = Dbase{1,1}+Dbase{2,2}; ss22 = ss11;
41 ii = [ii11; ii22]; jj = [jj11; jj22]; ss = [ss11; ss22];
42 A = sparse(ii,jj,ss,2*N,2*N);
43 A = mu*A;
44
45 % (div u,div v)
46 ss11 = Dbase{1,1}; ss12 = Dbase{1,2};
47 ss21 = Dbase{2,1}; ss22 = Dbase{2,2};
48 ii = [ii11; ii12; ii21; ii22];
49 jj = [jj11; jj12; jj21; jj22];
50 ss = [ss11; ss12; ss21; ss22];
51 B = sparse(ii,jj,ss,2*N,2*N);
52 B = (lambda+mu)*B;
53
54 % stiffness matrix
55 kk = A + B;
56
57 %% Assemble load vector
58 % Gauss quadrature rule
59 [lambda,weight] = quadpts(2);
60 F1 = zeros(NT,3); F2 = zeros(NT,3);
61 for p = 1:length(weight)
62     pxy = lambda(p,1)*node(elem(:,1),:) ...
63         + lambda(p,2)*node(elem(:,2),:) ...
64         + lambda(p,3)*node(elem(:,3),:);
65     fxy = f(pxy); % fxy = [f1xy,f2xy]
66     F1 = F1 + weight(p)*fxy(:,1)*lambda(p,:);
67     F2 = F2 + weight(p)*fxy(:,2)*lambda(p,:);
68 end
69 F1 = repmat(area,1,3).*F1; % F = area.*F;
70 F2 = repmat(area,1,3).*F2;
71 ff = accumarray([elem(:); elem(:)+N], [F1(:); F2(:)], [2*N 1]);
72
73 %% Apply Dirichlet boundary conditions
74 g_D = pde.g_D; bdNodeIdxD = bdStruct.bdNodeIdxD;
75 id = [bdNodeIdxD; bdNodeIdxD+N];
76 isBdNode = false(2*N,1); isBdNode(id) = true;
77 bdDof = (isBdNode); freeDof = (~isBdNode);
78 pD = node(bdNodeIdxD,:);
79 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
80 ff = ff - kk*u;
81
82 %% Set solver
83 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

6.5 线弹性问题的无闭锁有限元方法

P1 协调元求解线弹性问题会出现闭锁现象, 即当 Lamé 常数 λ 趋向无穷大时, 数值解不收敛. 其本质原因是, 当 $\lambda \rightarrow \infty$ 时, 线弹性问题的先验估计会导出 $\operatorname{div} \mathbf{u} = 0$ (Dirichlet 边界值为零). 但满足 $\operatorname{div} \mathbf{v}_h = 0$ 的 P1 协调元的函数只能是 $\mathbf{0}$. 克服闭锁现象的一种方法是使用非协调元.

6.5.1 Navier 形式的变分问题

Brenner 和 Sung 在 [5] 中证明了, Crouzeix-Raviart 非协调元对 Navier 形式是收敛的.

该非协调元的程序比较简单, 只要把基函数导数的配对乘以 4, 其他的修改都比较容易. 函数文件如下

CODE 6.5. elasticityNavierCR.m

```
1 function u = elasticityNavierCR(node, elem, pde, bdStruct)
2 % elasticityNavierCR solves linear elasticity equation of Navier form using ...
3 % Crouzeix-Raviart element
4 %
5 % u = [u1, u2]
6 % -mu \Delta u - (lambda + mu)*grad(div(u)) = f in \Omega
7 % Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D.
8 %
9 % Copyright (C) Terence Yu.
10 %
11 %% elem2dof of ui
12 allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
13 totalEdge = sort(allEdge,2);
14 [edge, ~, totalJ] = unique(totalEdge, 'rows');
15 NT = size(elem,1);
16 elem2edge = reshape(totalJ,NT,3);
17 elem2dof = elem2edge;
18 %
19 NE = size(edge,1);
20 Ndof = 3; % local dof number of ui
21 %
22 mu = pde.mu; lambda = pde.lambda; f = pde.f;
23 %
24 %% Compute (Dbase,Djbase)
25 [Dphi,area] = gradbasis(node, elem);
26 Dbase = cell(2,2);
27 for i = 1:2
28     for j = 1:2
29         k11 = 4*Dphi(:,i,1).*Dphi(:,j,1).*area;
30         k12 = 4*Dphi(:,i,1).*Dphi(:,j,2).*area;
31         k13 = 4*Dphi(:,i,1).*Dphi(:,j,3).*area;
```

```

31      k21 = 4*Dphi(:,i,2).*Dphi(:,j,1).*area;
32      k22 = 4*Dphi(:,i,2).*Dphi(:,j,2).*area;
33      k23 = 4*Dphi(:,i,2).*Dphi(:,j,3).*area;
34      k31 = 4*Dphi(:,i,3).*Dphi(:,j,1).*area;
35      k32 = 4*Dphi(:,i,3).*Dphi(:,j,2).*area;
36      k33 = 4*Dphi(:,i,3).*Dphi(:,j,3).*area;
37      K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in rows
38      Dbase{i,j} = K(:); % straighten
39
40 end
41
42 %% Get sparse assembling index
43 ii = reshape(repmat(elem2dof, Ndof,1), [], 1);
44 jj = repmat(elem2dof(:), Ndof, 1);
45 ii11 = ii;    jj11 = jj;   ii12 = ii;    jj12 = jj+NE;
46 ii21 = ii+NE; jj21 = jj;   ii22 = ii+NE; jj22 = jj+NE;
47
48 %% Assemble stiffness matrix
49 % (grad u,grad v)
50 ss11 = Dbase{1,1}+Dbase{2,2};  ss22 = ss11;
51 ii = [ii11; ii22]; jj = [jj11; jj22]; ss = [ss11; ss22];
52 A = sparse(ii,jj,ss,2*NE,2*NE);
53 A = mu*A;
54
55 % (div u,div v)
56 ss11 = Dbase{1,1};           ss12 = Dbase{1,2};
57 ss21 = Dbase{2,1};           ss22 = Dbase{2,2};
58 ii = [ii11; ii12; ii21; ii22];
59 jj = [jj11; jj12; jj21; jj22];
60 ss = [ss11; ss12; ss21; ss22];
61 B = sparse(ii,jj,ss,2*NE,2*NE);
62 B = (lambda+mu)*B;
63
64 % stiffness matrix
65 kk = A + B;
66
67 %% Assemble load vector
68 % Gauss quadrature rule
69 [lambda,weight] = quadpts(3);
70 F1 = zeros(NT,3); F2 = zeros(NT,3);
71 phi = 1-2*lambda; % basis functions for CR element
72 for p = 1:length(weight)
73     pxy = lambda(p,1)*node(elem(:,1),:) ...
74         + lambda(p,2)*node(elem(:,2),:) ...
75         + lambda(p,3)*node(elem(:,3),:);
76     fxy = f(pxy); % fxy = [f1xy,f2xy]
77     F1 = F1 + weight(p)*fxy(:,1)*phi(p,:);
78     F2 = F2 + weight(p)*fxy(:,2)*phi(p,:);
79 end
80 F1 = repmat(area,1,3).*F1; % F = area.*F;

```

```

81 F2 = repmat(area,1,3).*F2;
82 ff = accumarray([elem2dof(:); elem2dof(:)+NE], [F1(:); F2(:)], [2*NE 1]);
83
84 %% Apply Dirichlet boundary conditions
85 isBdDof = false(2*NE,1);
86 bdEdgeIdxD = bdStruct.bdEdgeIdxD;
87 fixedNode = [bdEdgeIdxD; bdEdgeIdxD+NE];
88 isBdDof(fixedNode) = true;
89 bdDof = (isBdDof); freeDof = (~isBdDof);
90 bdEdgeD = bdStruct.bdEdgeD;
91 zm = (node(bdEdgeD(:,1),:)+node(bdEdgeD(:,2),:))/2;
92 u = zeros(2*NE,1); u(bdDof) = pde.g_D(zm);
93 ff = ff - kk*u;
94
95 %% Set up solver type
96 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

$\lambda = 10^8, \mu = 1$ 的结果见下图.

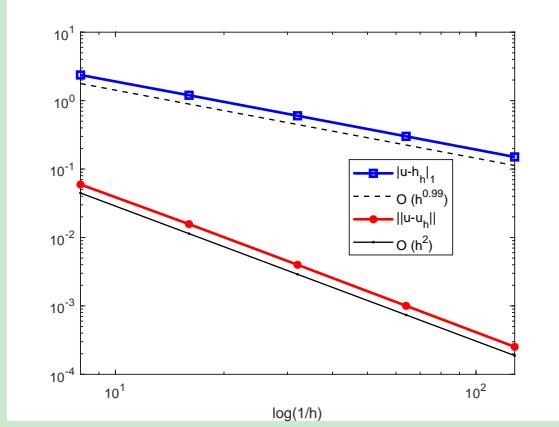


图 6.1. 线弹性问题 Navier 形式 CR 元的收敛性

6.5.2 Traction 形式的变分问题

分片一次多项式近似不可以用来直接求解第二种变分形式 (6.7) 对应的问题, 这是因为此时不成立离散情形的 Korn 不等式 (例如见文献 [6] 的 Sect. 6). 实际上, 对前面的例子, 编程发现 Crouzeix-Raviart 元不收敛.

当考虑纯粹 Traction 问题

$$\begin{cases} -\operatorname{div} \boldsymbol{\sigma} = \mathbf{f} & \text{in } \Omega, \\ \boldsymbol{\sigma} \mathbf{n} = \mathbf{g} & \text{on } \partial\Omega \end{cases}$$

时, 合适的函数空间为

$$\widehat{\mathbf{H}}^k(\Omega) = \left\{ \mathbf{v} \in \mathbf{H}^k(\Omega) : \int_{\Omega} \mathbf{v} dx = \mathbf{0}, \quad \int_{\Omega} \nabla \times \mathbf{u} dx = 0 \right\},$$

它是 $\mathbf{H}^k(\Omega)$ 的闭子空间, 且在该空间中成立 Korn 不等式. 这意味着连续变分问题的解在该空间中. 另外,

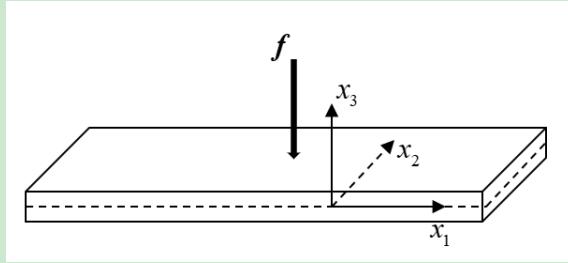
$$\mathbf{H}^k(\Omega) = \widehat{\mathbf{H}}^k(\Omega) + \mathbf{RM}(\Omega),$$

这里的和不是直和.

第七章 Kirchhoff 板弯问题

7.1 变分问题

7.1.1 平衡方程与边界条件



给定一个薄板, 即厚度 t 远小于水平方向的尺度. 称板厚方向为横向, 水平方向为纵向. 在板厚方向施加横向的载荷, 薄板会发生弯曲 (小变形), 且设板厚中面的位移, 即挠度为 w .

在弹性力学基本假设以及 Kirchhoff 计算假设下, 薄板弯曲问题可简化为挠度 w 的平面问题, 平衡方程为

$$\Omega : \quad -\partial_{ij} M_{ij}(w) = f, \quad (7.1)$$

式中,

$$M_{ij} = D ((1 - \nu) K_{ij} + \nu K_{kk} \delta_{ij}), \quad K_{ij} = -\partial_{ij} w,$$

指标范围为 $i, j = 1, 2, k \in \{1, 2\}$. 该方程是挠度 w 的四阶椭圆型偏微分方程. 若板面与地基有弹性耦合, 则平衡方程可改写为

$$\Omega : \quad -\partial_{ij} M_{ij}(w) + cw = f,$$

其中 c 为弹性耦合常数.

当 D, ν 为常数时, 含有 ν 的项会抵消, 上述方程简化为双调和方程

$$D\Delta^2 w = f.$$

注意, 对任意的 ν 都是如此, 特别地, 取 $\nu = 0$ (工程中是不允许的), 原来的平衡方程就是 $D\partial_{ij}\partial_{ij} w = f$, 显然 $\partial_{ij}\partial_{ij} = \Delta^2$. 为了方便, 以下将坐标 (x_1, x_2) 改记为 (x, y) .

本文考虑强加边界条件

$$w = \partial_n w = 0, \quad (x, y) \in \partial\Omega.$$

7.1.2 变分问题

在平衡方程 (7.1) 两边乘以检验函数 v 并分部积分后可获得变分问题, 此时会出现复杂的边界项. 但当 $v \in H_0^2(\Omega)$ 时, 这些边界项都会消失.

变分问题为: 找 $u \in V := H_0^2(\Omega)$ 使得

$$D(w, v) = F(v), \quad v \in V,$$

式中,

$$\begin{aligned} D(w, v) &= \int_{\Omega} M_{ij}(w) K_{ij}(v) dx dy + \int_{\Omega} c w v dx dy, \\ F(v) &= \int_{\Omega} f v dx dy. \end{aligned}$$

命

$$M = \begin{bmatrix} M_{11} \\ M_{22} \\ M_{12} \end{bmatrix}, \quad K = \begin{bmatrix} K_{11} \\ K_{22} \\ 2K_{12} \end{bmatrix},$$

则

$$\int_{\Omega} M_{ij}(w) K_{ij}(v) dx dy = \int_{\Omega} K^T(v) M(w) dx dy,$$

且薄板弯曲的 Hooke 定律可写为

$$M = R K, \quad R = D \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & (1-\nu)/2 \end{bmatrix}. \quad (7.2)$$

注意, 我们有

$$\begin{aligned} \int_{\beta} M_{ij}(w) K_{ij}(v) dx dy &= D \int_{\beta} [\partial_{11} w \partial_{11} v + \partial_{22} w \partial_{22} v \\ &\quad + \nu (\partial_{11} w \partial_{22} v + \partial_{22} w \partial_{11} v) \\ &\quad + 2(1-\nu) \partial_{12} w \partial_{12} v] dx dy. \end{aligned} \quad (7.3)$$

7.1.3 有限元方法

假设单元上有 6 个自由度, 相应的节点变量和节点基函数记为

$$W_{\beta} = [w_1, \dots, w_6]^T, \quad N = [\varphi_1, \dots, \varphi_6],$$

注意基函数采用行向量写法 (回忆线性代数), 则插值函数 Q 可写为

$$Q = NW_{\beta}.$$

由 (7.2), 只需要给出 K 的近似. 显然有

$$K = \begin{bmatrix} K_{11} \\ K_{22} \\ 2K_{12} \end{bmatrix} = - \begin{bmatrix} \partial_{11}w \\ \partial_{22}w \\ 2\partial_{12}w \end{bmatrix} \approx - \begin{bmatrix} \partial_{11}Q \\ \partial_{22}Q \\ 2\partial_{12}Q \end{bmatrix} = - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} W_\beta =: BW_\beta,$$

其中,

$$B = - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} = [B_1, \dots, B_6], \quad B_i = - \begin{bmatrix} \partial_{11}\varphi_i \\ \partial_{22}\varphi_i \\ 2\partial_{12}\varphi_i \end{bmatrix}.$$

双线性形式的第一项为

$$\begin{aligned} & \int_\beta M_{ij}(w) K_{ij}(v) dx dy \\ &= \int_\beta K^T(v) M(w) dx dy = \int_\beta K^T(v) R K(w) dx dy \\ &= V_\beta^T \int_\beta B^T R B dx dy W_\beta =: V_\beta^T K_\beta W_\beta, \end{aligned}$$

式中,

$$\begin{aligned} K_\beta &= \int_\beta B^T R B dx dy = (K_{st})_{6 \times 6}, \\ K_{st} &= \int_\beta B_s^T R B_t dx dy =: (k_{ij}), \end{aligned}$$

满足

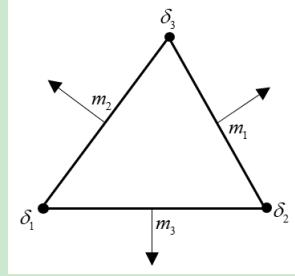
$$\begin{aligned} k_{ij} &= D \int_\beta [\partial_{11}\varphi_i \partial_{11}\varphi_j + \partial_{22}\varphi_i \partial_{22}\varphi_j \\ &\quad + \nu(\partial_{11}\varphi_i \partial_{22}\varphi_j + \partial_{22}\varphi_i \partial_{11}\varphi_j) \\ &\quad + 2(1-\nu)\partial_{12}\varphi_i \partial_{12}\varphi_j] dx dy. \end{aligned} \tag{7.4}$$

7.2 非协调 Morley 元

7.2.1 Morley 元的构造

Morley 元也称为完全二次三角形元, 它连 C^0 连续都不是.

局部节点基



Morley 元的三元组 $(K, \mathcal{P}, \mathcal{N})$ 中的

$K = \beta$ 为三角形,

$\mathcal{P} = \mathbb{P}_2(K)$,

$\mathcal{N} = \{v(p), p = \delta_1, \delta_2, \delta_3; \partial_n v(m), m = m_1, m_2, m_3\}$.

注 7.1 注意 $\partial_n v = \nabla v \cdot \vec{n}$, 对相邻两个单元来说, $\partial_n v(m_i)$ 是不同的. 事实上, 它跨边界仍是连续的, 从而

$$\partial_n v(m_i)|_{\beta_1} = -\partial_n v(m_i)|_{\beta_2},$$

其中, m_i 是相邻三角形 β_1 和 β_2 公共边的中点. 对该点来说, 整体自由度只要取定一个方向即可, 注意局部基函数会相差符号, 后面说明.

注 7.2 边中点法向导数也可换为如下的积分平均

$$\frac{1}{|e_i|} \int_{e_i} \partial_n v ds,$$

此时基函数和有限元空间不变 (仍有方向性问题).

现在考虑局部节点基. 取三角形 $\beta = (\delta_1, \delta_2, \delta_3)$, 三对边的中点分别记为 m_1, m_2, m_3 . 以这 6 个点作为插值点, 构造一个完全二次多项式

$$Q(x, y) = a_1 + a_2x + a_3y + a_4x^2 + a_5xy + a_6y^2,$$

要求在顶点上的函数值以及三边中点上法向导数值相同, 即

$$Q(\delta_i) = w_i, \quad i = 1, 2, 3; \quad Q_n(m_i) = w_{mi}, \quad i = 1, 2, 3,$$

这 6 个条件可唯一确定多项式的 6 个系数. 设节点基函数分别为 $\varphi_i, \psi_i(x, y)$, $i = 1, 2, 3$, 即

$$Q(x, y) = \sum_{i=1}^3 w_i \varphi_i(x, y) + \sum_{i=1}^3 w_{mi} \psi_i(x, y), \quad (7.5)$$

满足

- 对 $i = 1, 2, 3$, 有

$$\varphi_i(\delta_j) = \delta_{ij}, \quad j = 1, 2, 3; \quad \partial_n \varphi_i(m_j) = 0, \quad j = 1, 2, 3.$$

- 对 $i = 4, 5, 6$, 有

$$\varphi_i(\delta_j) = 0, \quad j = 1, 2, 3; \quad \partial_n \varphi_i(m_j) = \delta_{ij}, \quad j = 1, 2, 3.$$

由这些条件, 可用重心坐标来构造基函数 φ_i . 设重心坐标为 $\lambda_1, \lambda_2, \lambda_3$, i, j, k 表示 1, 2, 3 的轮换. 定义

$$\xi_i = x_j - x_k, \quad \eta_i = y_j - y_k, \quad \omega_i = x_j y_k - x_k y_j,$$

则三角形的有向面积可表示为

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} = \frac{1}{2} (\xi_1 \eta_2 - \xi_2 \eta_1) = \omega_1 \omega_2 \omega_3,$$

且

$$\lambda_i(x, y) = \frac{1}{2S} (\eta_i x - \xi_i y + \omega_i), \quad i = 1, 2, 3. \quad (7.6)$$

从 (x, y) 到 (λ_1, λ_2) 变换的 Jacobi 矩阵的行列式为

$$|J| = \det \frac{\partial(\lambda_1, \lambda_2)}{\partial(x, y)} = \frac{1}{2S}.$$

再令

$$s_{ij} = -(\xi_i \xi_j + \eta_i \eta_j),$$

则有

$$\begin{cases} \varphi_1 = \lambda_1^2 + \left(\frac{s_{12}}{l_2^2} + \frac{s_{31}}{l_3^2} \right) \lambda_2 \lambda_3 + \frac{s_{31}}{l_3^2} \lambda_3 \lambda_1 + \frac{s_{12}}{l_2^2} \lambda_1 \lambda_2, \\ \varphi_2 = \lambda_2^2 + \frac{s_{23}}{l_3^2} \lambda_2 \lambda_3 + \left(\frac{s_{23}}{l_3^2} + \frac{s_{12}}{l_1^2} \right) \lambda_3 \lambda_1 + \frac{s_{12}}{l_1^2} \lambda_1 \lambda_2, \\ \varphi_3 = \lambda_3^2 + \frac{s_{23}}{l_2^2} \lambda_2 \lambda_3 + \frac{s_{31}}{l_1^2} \lambda_3 \lambda_1 + \left(\frac{s_{31}}{l_1^2} + \frac{s_{23}}{l_2^2} \right) \lambda_1 \lambda_2, \\ \psi_1 = \frac{2S}{l_1} \lambda_1 (\lambda_1 - 1), \quad \psi_2 = \frac{2S}{l_2} \lambda_2 (\lambda_2 - 1), \quad \psi_3 = \frac{2S}{l_3} \lambda_3 (\lambda_3 - 1), \end{cases}$$

式中, l_i 表示第 i 个顶点所对边的边长 (注意基函数中指标的轮换).

整体节点基

前面已经指出了边中点自由度的方向性问题. 不考虑边界条件, Morley 元的整体有限元空间为

$$V_h = \{v \in L^2(\Omega) : v \text{ 的 Morley 元自由度连续}\}.$$

对内部边 e , 可任意取定一个三角形的外法向量参数作为整体节点参数, 不妨称其为定参三角形. 此时, 整体节点基限制在定参三角形上就是局部节点基, 而限制在相邻三角形上则与局部节点基相差负号.

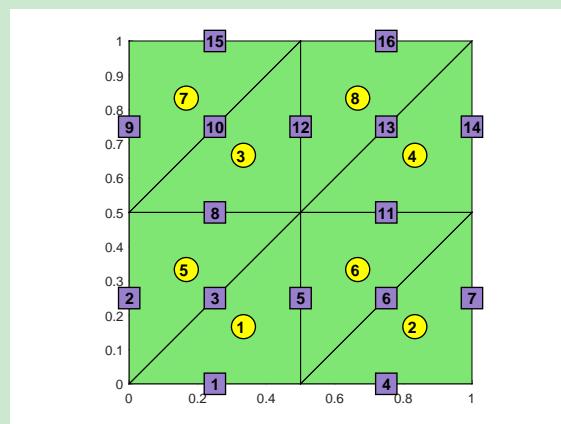
7.2.2 自由度的方向处理

方法一: 边的符号化

从基函数中可以看到, ψ_i 中的 l_i 可起到变号的作用, 即可规定 l_i 是有向长度 (此时 φ 不会受到影响).

1. 边的定向: 对三角形单元的边 e , 规定逆时针方向为其在单元中的正向, 可用局部编号来确定.
2. 边的符号:
 - 对内部定向边, 称终点与起点整体编号差的符号为该边在单元中的符号.
 - 对边界边, 规定符号为正 (即 +1).
 - 显然对内部边, 限制在相邻两个单元上的符号相异.

下面说明如何获得边的符号, 用矩阵 sgn_{elem} 存储, 它是按单元给出的.



- 辅助数据结构中给出了 elem2edge , 它按单元存储每条边的自然序号 (符合第 i 个顶点对边为第 i 条边), 而 edge 存储了一维边的端点编号. 可如下计算按单元存储的边的长度

```

1 % edge length
2 z1 = node(edge(:,1),:); z2 = node(edge(:,2),:);
3 he = sqrt(sum((z2-z1).^2,2)); L = he(elem2edge);

```

- 现在给边添上符号. 执行如下语句可获得所有单元内部边的符号

```

1 sgnelem = sign([elem(:,3)-elem(:,2), elem(:,1)-elem(:,3), elem(:,2)-elem(:,1)]);

```

这里边界边的符号可能不正确.

- 利用逻辑运算可修改 `sgnelem` 中边界边的符号差, 从而获得带符号的边长 `sgnL`.

```

1 bdEdgeIdx = bdStruct.bdEdgeIdx;
2 E = false(NE,1); E(bdEdgeIdx) = 1; sgnbd = E(elem2edge);
3 sgnelem(sgnbd) = 1;
4 sgnL = sgnelem.*L;

```

这里, `bdEdgeIdx` 是边界边的自然序号, `E` 内部边对应 `false`, 边界边对应 `true`.

这种方法可直接进行装配 (见装配的说明)

```

1 ii = reshape(repmat(elem2dof, Ndof, 1), [], 1);
2 jj = repmat(elem2dof(:, ), Ndof, 1);
3 kk = sparse(ii,jj,K(:, )+G(:, ),NNdof,NNdof);
4 ff = accumarray(elem2dof(:, ), F(:, ), [NNdof 1]);

```

方法二: 符号刚度矩阵和符号载荷向量

前面的处理方法不具有一般性. 一方面, 它需要知道基函数的具体形式, 一方面, 它改变单元刚度矩阵. 边进行符号化本质上就是基函数的符号化. 注意到

$$a_K(\pm\varphi_i, \pm\varphi_j) = \pm \cdot \pm a_K(\varphi_i, \varphi_j),$$

为此可对应给出一个符号矩阵, 它记录 (i, j) 位置的符号. 如下获得

```

1 sgnbase = ones(NT,Ndof); sgnbase(:,4:6) = sgnelem;
2 sgnK = zeros(NT,Ndof^2);
3 s = 1;
4 for i = 1:Ndof
5     for j = 1:Ndof
6         sgnK(:,s) = sgnbase(:,i).*sgnbase(:,j);
7         s = s+1;
8     end
9 end

```

这里, `sgnbase` 记录全局基函数限制后的符号, 显然顶点对应的基函数符号都为 1, 而中点的恰好对应 `sgnelem`. 可以看到, 它与单元刚度矩阵的获取类似. 同理, 可给出符号载荷向量.

```
1 sgnF = ones(NT,Ndof); sgnF(:,4:6) = sgnelem; % sign vector
```

装配时只要把刚度矩阵和载荷向量分别点乘符号刚度矩阵和符号载荷向量即可, 如下 (见后面说明)

```
1 kk = sparse(ii,jj,(K(:)+G(:)).*sgnK(:,N+NE,N+NE));
2 ff = accumarray(elem2(:), F(:).*sgnF(:, [N+NE 1]));
```

显然这种处理可推广至任意涉及方向的自由度问题以及多角形剖分的问题 (Morley 元的虚拟方法).

后面对第二种方法进行说明.

7.2.3 刚度矩阵与载荷向量的计算

双线性形式第一项的计算

基函数的二阶导数为常数, 由 (7.4),

$$K_{ij} = \int_{\beta} B_i^T R B_j dxdy = |\beta| B_i^T R B_j = |\beta| D \cdot (k_{ij})_{6 \times 6},$$

其中,

$$k_{ij} = \partial_{11}\varphi_i \partial_{11}\varphi_j + \partial_{22}\varphi_i \partial_{22}\varphi_j + \nu(\partial_{11}\varphi_i \partial_{22}\varphi_j + \partial_{22}\varphi_i \partial_{11}\varphi_j) + 2(1-\nu)\partial_{12}\varphi_i \partial_{12}\varphi_j.$$

直接计算有

$$\begin{cases} \partial_{11}\varphi_1 = \frac{1}{2S^2} \left(\eta_1^2 - \frac{s_{12}}{l_2^2} \eta_2^2 - \frac{s_{31}}{l_3^2} \eta_3^2 \right), \\ \partial_{22}\varphi_1 = \frac{1}{2S^2} \left(\xi_1^2 - \frac{s_{12}}{l_2^2} \xi_2^2 - \frac{s_{31}}{l_3^2} \xi_3^2 \right), \\ \partial_{12}\varphi_1 = -\frac{1}{2S^2} \left(\xi_1 \eta_1 - \frac{s_{12}}{l_2^2} \xi_2 \eta_2 - \frac{s_{31}}{l_3^2} \xi_3 \eta_3 \right), \end{cases}$$

对 φ_2 的各阶导数, 在上式中把右端的 $(1, 2, 3) \rightarrow (2, 3, 1)$; 对 φ_3 的各阶导数, 在上式中把右端的 $(1, 2, 3) \rightarrow (3, 1, 2)$. 而

$$\begin{cases} \partial_{11}\varphi_{3+i} = \frac{1}{Sl_i} \eta_i^2 \\ \partial_{22}\varphi_{3+i} = \frac{1}{Sl_i} \xi_i^2 \\ \partial_{12}\varphi_{3+i} = -\frac{1}{Sl_i} \xi_i \eta_i \end{cases}, \quad i = 1, 2, 3.$$

1. 计算中用到的 ξ_i, η_i 如下获得

```
1 % area
2 z1 = node(elem(:,1),:);
3 z2 = node(elem(:,2),:);
4 z3 = node(elem(:,3),:);
5 e1 = z2-z3; e2 = z3-z1; e3 = z1-z2; % ei = [xi, eta]
6 area = 0.5*(-e3(:,1).*e2(:,2)+e3(:,2).*e2(:,1));
7 % xi, eta
8 xi = [e1(:,1), e2(:,1), e3(:,1)];
9 eta = [e1(:,2), e2(:,2), e3(:,2)];
```

所有单元的 s_{ij} 用三维数组存储

```
1 % sij
2 sb = zeros(NT,3,3);
3 for i = 1:3
4     for j = 1:3
5         sb(:,i,j) = -xi(:,i).*xi(:,j) - eta(:,i).*eta(:,j);
6     end
7 end
```

也可用元胞数组存储, 但不利于向量化运算.

2. 现在计算基函数的二阶导数. 我们用 b_{11} 存储所有基函数的 ∂_{11} 导数, 即

$$b_{11} = [\partial_{11}\varphi_1, \partial_{11}\varphi_2, \dots, \partial_{11}\varphi_6],$$

且每行对应一个单元. 基函数中的系数如下生成.

```
1 % coefficients in the basis functions
2 ind = [1 2 3; 2 3 1; 3 1 2]; % rotation index
3 it = ind(:,1); jt = ind(:,2); kt = ind(:,3);
4 c0 = zeros(NT,3); c1 = c0; c2 = c0;
5 for i = 1:3
6     j = ind(i,2); k = ind(i,3);
7     c0(:,i) = 1./(2*area.^2);
8     c1(:,i) = sb(:,i,j)./L(:,j).^2;
9     c2(:,i) = sb(:,k,i)./L(:,k).^2;
10 end
11 c3 = c1+c2;
```

基函数的二阶导为

```
1 %% Second derivatives of basis functions
2 b11 = zeros(NT,6); b22 = b11; b12 = b11; % [phi_i, i=1:6]
3 % i = 1,2,3
4 b11(:,it) = c0.* (eta(:,it).^2 - c1.*eta(:,jt).^2 - c2.*eta(:,kt).^2);
5 b22(:,it) = c0.* (xi(:,it).^2 - c1.*xi(:,jt).^2 - c2.*xi(:,kt).^2);
```

```

6 b12(:,it) = -c0.* (xi(:,it).*eta(:,it) - c1.*xi(:,jt).*eta(:,jt) - ...
7 % i = 4,5,6
8 ci = 1./(repmat(area,1,3).*L(:,it));
9 b11(:,3+it) = ci.*eta(:,it).^2;
10 b22(:,3+it) = ci.*xi(:,it).^2;
11 b12(:,3+it) = -ci.*xi(:,it).*eta(:,it);

```

注意这里用指标的轮换进行计算.

3. 第一项对应的单元刚度矩阵为

```

1 %% First stiffness matrix and sign matrix
2 K = zeros(NT,Ndof^2); sgnK = zeros(NT,Ndof^2);
3 s = 1;
4 for i = 1:Ndof
5     for j = 1:Ndof
6         K(:,s) = b11(:,i).*b11(:,j) + b22(:,i).*b22(:,j) ...
7             + nu*(b11(:,i).*b22(:,j) + b22(:,i).*b11(:,j)) ...
8             + 2*(1-nu)*b12(:,i).*b12(:,j);
9         sgnK(:,s) = sgnbase(:,i).*sgnbase(:,j);
10        s = s+1;
11    end
12 end
13 K = D*repmat(area,1,Ndof^2).*K;

```

双线性形式第二项的计算

双线性形式的第二项为

$$\int_{\beta} c w v dxdy = V_{\beta}^T \int_{\beta} c N^T N dxdy W_{\beta} = V_{\beta}^T G_{\beta} W_{\beta},$$

其中地基能量阵

$$G_{\beta} = \int_{\beta} c N^T N dxdy = (g_{ij})_{6 \times 6}, \quad g_{ij} = \int_{\beta} c \varphi_i \varphi_j dxdy.$$

以下用三角形上的 Gauss 公式计算, 参考前面的说明, 如节 5.2.2.

(a) 我们要对积分点进行循环, 如下可获得所有单元固定积分点处的基函数值.

```

1 % basis functions at the p-th quadrature point
2 base = zeros(NT,Ndof);
3 % i = 1,2,3
4 base(:,it) = repmat(lambda(p,it).^2,NT,1) ...
5     + c3.*repmat(lambda(p,jt).*lambda(p,kt),NT,1) ...
6     + c2.*repmat(lambda(p,kt).*lambda(p,it),NT,1) ...
7     + c1.*repmat(lambda(p,it).*lambda(p,jt),NT,1);

```

```

8      % i = 4,5,6
9      ci = 2*repmat(area,1,3)./L(:,it);
10     base(:,3+it) = ci.*repmat(lambda(p,it).*lambda(it)-1),NT,1);

```

(b) 刚度矩阵 G 如下计算

```

1  % Second stiffness matrix
2  s = 1;
3  for i = 1:Ndof
4      for j = 1:Ndof
5          gs = cf(pxy).*base(:,i).*base(:,j);
6          G(:,s) = G(:,s) + weight(p)*gs;
7          s = s+1;
8      end
9  end

```

这里只是第 p 个积分点的部分, 循环即得最终的矩阵 (未乘以单元面积). 系数函数若是常数, 可如下转化为匿名函数

```

1 if isnumeric(para.c), cf = @(xy) para.c+0*xy(:,1); end

```

循环即可获得 G .

注 7.3 当 $c = 0$ 时, 上面的一些计算可以去掉, 但其过程在载荷向量的计算中用到.

载荷向量的计算

右端

$$\int_{\beta} f v dx dy = V_{\beta}^T \int_{\beta} f N^T dx dy = V_{\beta}^T F_{\beta},$$

其中,

$$F_{\beta} = \int_{\beta} f N dx dy = (F_i)_{6 \times 1}, \quad F_i = \int_{\beta} f \varphi_i dx dy.$$

载荷向量可如下计算

```

1 F = F + weight(p)*repmat(f(pxy),1,Ndof).*base;

```

循环即得最终的 (未乘以单元面积).

7.2.4 边界条件的处理

我们考虑的是强加边界条件

$$w = \partial_n w = 0, \quad (x, y) \in \partial\Omega,$$

这里可以是非齐次的 (这两个条件都是 Dirichlet 边界条件).

$w|_{\partial\Omega} = g$ 与常规情形一样处理. 现考虑 $\partial_n w|_{\partial\Omega} = g_n$.

1. Morley 元构造时, 三角形边上中点处的法向导数值是自由度. 这样, 第二个条件可直接替换已知法向导数值, 这与第一个条件是类似的.
2. 在 setboundary.m 中我们给出了 Dirichlet 点的编号 `bdNodeIdx` 以及边界边的编号 `bdEdgeD`, 这样约束点的编号为

```
1 bdNodeIdx = bdStruct.bdNodeIdx;
2 bdEdgeD = bdStruct.bdEdgeD;
3 fixedNode = [bdNodeIdx; bdEdgeD+N];
```

由此给出自由点编号

```
1 isBdNode = false(NN dof, 1); isBdNode(fixedNode) = true;
2 freeDof = ~isBdNode;
```

3. 边界节点值为

```
1 g_D = pde.g_D;
2 nodeD = node(eD,:); wD = g_D(nodeD);
```

边界边中点法向导数值如下计算

```
1 Dw = pde.Dw;
2 z1 = node(bdEdgeD(:,1),:);
3 z2 = node(bdEdgeD(:,2),:);
4 zc = (z1+z2)./2;
5 e = z1-z2; % e = z2-z1
6 ne = [-e(:,2), e(:,1)]; % scaled
7 ne = ne./repmat(he(bdEdgeIdx), 1, 2);
8 wND = sum(Dw(zc).*ne, 2);
```

这里 `Dw` 是梯度 ∇w , 而 $\partial_n w = \nabla w \cdot \vec{n}$ 中的 \vec{n} 通过边界边的旋转获得 (并进行单位化).

4. 这样, 我们如下求解

```
1 w = zeros(N+NE, 1); w(bdDof) = [wD; wND];
2 ff = ff - kk*w;
3 w(freeDof) = kk(freeDof, freeDof)\ff(freeDof);
```

7.2.5 数值结果

主程序为 `main_PlateBendingMorley.m`, 相应的函数文件为 `PlateBendingMorley.m`, 用

到的数据文件为 PlateBendingData.m. 这里不再给出具体程序, 参见 GitHub 上传程序 (elasticity 文件夹中).

区域 $[0, 1]^2$ 的结果如下

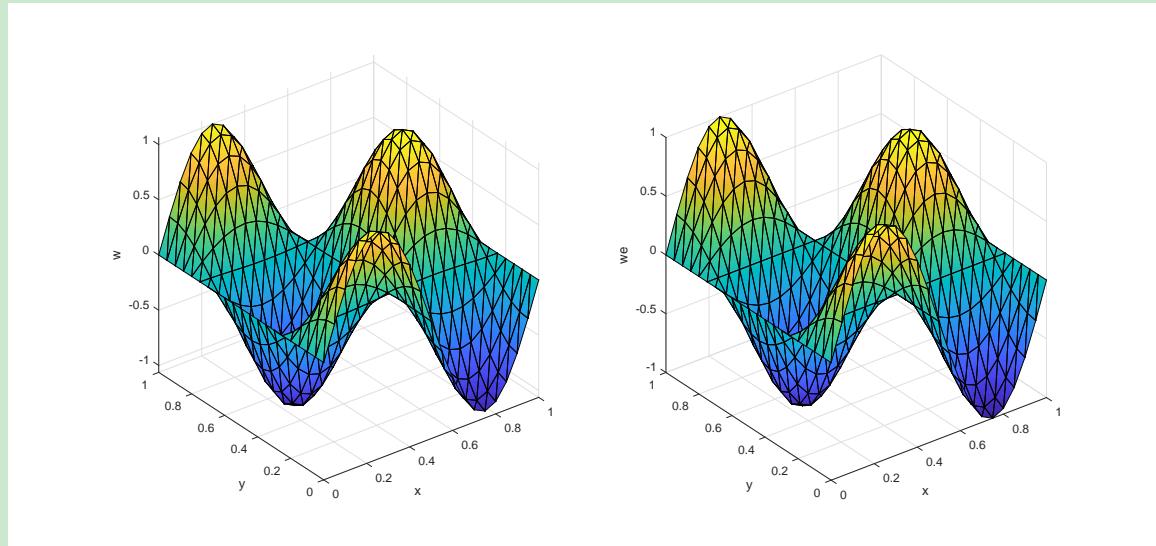


图 7.1. Morley 元方法的数值解与精确解 ($N_x=N_y=20$)

绝对误差见下图

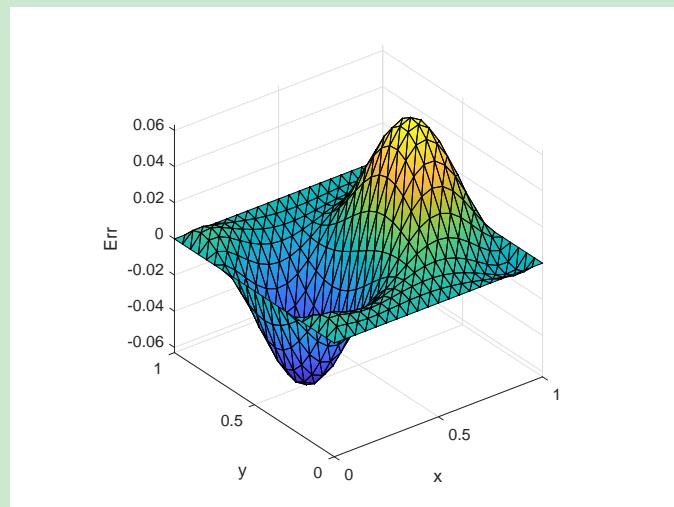


图 7.2. Morley 元方法的绝对误差 ($N_x=N_y=20$)

7.3 非协调 Zienkiewicz 元

Zienkiewicz 元是一种不完全三次三角形元 (Serendipity element), 它是 C^0 连续的.

7.3.1 Zienkiewicz 元的构造

三次插值多项式有 10 个自由度, 一种减少自由度的方法就是直接去掉三次多项式基中的某些项或者某些项共用一个系数.

1. 我们知道三次多项式在面积坐标下可表示成 λ_1, λ_2 和 λ_3 的齐三次式, 即 $\forall P \in \mathbb{P}_3$, 有 $P = \sum_{i+j+k=3} c_{ijk} \lambda_1^i \lambda_2^j \lambda_3^k$. 因形心对应的节点基为 $27\lambda_1\lambda_2\lambda_3$, 故一个例子是在 $P = \sum_{i+j+k=3} a_{ijk} \lambda_1^i \lambda_2^j \lambda_3^k$ 中去掉 $\lambda_1\lambda_2\lambda_3$ 的这一项构造不完全的三次插值.
2. 尽管在边上附加 9 个条件可唯一确定该多项式, 但这样的插值对一次式都不准确. 事实上, 对一次式 $P_1(\lambda) = \lambda_1$, 我们把它表为齐三次式如下

$$\begin{aligned} P_1(\lambda) &= \lambda_1 = \lambda_1(\lambda_1 + \lambda_2 + \lambda_3)^2 \\ &= \lambda_1^3 + \lambda_1\lambda_2^2 + \lambda_1\lambda_3^2 + 2\lambda_1^2\lambda_2 + 2\lambda_1^2\lambda_3 + 2\lambda_1\lambda_2\lambda_3. \end{aligned}$$

这表明为了让一次式精确, 不仅 $\lambda_1\lambda_2\lambda_3$ 不能丢掉, 而且所有含 λ_1 的齐三次单项式都不能去掉. 同理为了让 $P_i(\lambda) = \lambda_i$ 精确, 所有含有 λ_i 的齐三次单项式都不能去掉.

3. 我们的目标是插值的逼近程度尽量高. 三次多项式已不可能, 只好退而求其次, 考虑逼近程度达到二次多项式. 也就是说, 在已有的 9 个自由度的前提下, 再加一个限制条件: 插值的逼近程度达到二次多项式. 由这个限制可获得插值多项式系数的新关系式 (见文献 [1] 中的 (6.25) 式).

给定插值多项式 Q , Zienkiewicz 元构造的条件是

- 给定三角形顶点处的函数值以及两个偏导数值, 即

$$Q(\delta_i) = w_i, \quad Q_x(\delta_i) = w_{xi}, \quad Q_y(\delta_i) = w_{yi}, \quad i = 1, 2, 3.$$

- 插值对二次多项式精确, 即插值多项式包含二次多项式.

经过一些细致的推导, 我们有

$$Q(x, y) = \sum_{i=1}^3 w_i \varphi_i + w_{xi} \psi_i + w_{yi} \zeta_i,$$

其中,

$$\begin{cases} \varphi_1 = \lambda_1^2(3 - 2\lambda_1) + 2\lambda_1\lambda_2\lambda_3, \\ \varphi_2 = \lambda_2^2(3 - 2\lambda_2) + 2\lambda_1\lambda_2\lambda_3, \\ \varphi_3 = \lambda_3^2(3 - 2\lambda_3) + 2\lambda_1\lambda_2\lambda_3, \end{cases}$$

$$\begin{cases} \psi_1 = \lambda_1^2(\xi_1\lambda_2 - \xi_2\lambda_1 + \xi_2) + (\frac{1}{2}\xi_1 + \xi_2)\lambda_1\lambda_2\lambda_3, \\ \psi_2 = \lambda_2^2(\xi_1\lambda_2 - \xi_2\lambda_1 - \xi_1) - (\frac{1}{2}\xi_2 + \xi_1)\lambda_1\lambda_2\lambda_3, \\ \psi_3 = \lambda_3^2(\xi_1\lambda_2 - \xi_2\lambda_1) + \frac{1}{2}(\xi_1 - \xi_2)\lambda_1\lambda_2\lambda_3, \end{cases}$$

$$\begin{cases} \zeta_1 = \lambda_1^2(\eta_1\lambda_2 - \eta_2\lambda_1 + \eta_2) + (\frac{1}{2}\eta_1 + \eta_2)\lambda_1\lambda_2\lambda_3, \\ \zeta_2 = \lambda_2^2(\eta_1\lambda_2 - \eta_2\lambda_1 - \eta_1) - (\frac{1}{2}\eta_2 + \eta_1)\lambda_1\lambda_2\lambda_3, \\ \zeta_3 = \lambda_3^2(\eta_1\lambda_2 - \eta_2\lambda_1) + \frac{1}{2}(\eta_1 - \eta_2)\lambda_1\lambda_2\lambda_3. \end{cases}$$

可以检验，这些基函数的系数满足限制条件。

7.3.2 sparse 装配指标

整体自由度如下排列

$$W = [w_1, \dots, w_N, w_{x1}, \dots, w_{xN}, w_{y1}, \dots, w_{yN}],$$

即先排列所有顶点函数值，再排顶点处对 x 求偏导的值，最后排对 y 求偏导的值。

文献 [1] 在分析单元的时候，将单元变量如下排列

$$[w_1, w_{x1}, w_{y1}, \quad w_2, w_{x2}, w_{y2}, \quad w_3, w_{x3}, w_{y3}]^T,$$

事实上，工程书上一般都是如此做的。但从编程的向量化角度来看，最好还是按照 Morley 元的情形如下排列

$$W_\beta = [w_1, w_2, w_3, \quad w_{x1}, w_{x2}, w_{x3}, \quad w_{y1}, w_{y2}, w_{y3}]^T.$$

显然装配指标为

```

1 N = size(node,1); NT = size(elem,1); Ndof = 9;
2 elem2dof = [elem, elem+N, elem+2*N];
3 ii = reshape(repmat(elem2dof, Ndof,1), [], 1);
4 jj = repmat(elem2dof(:, ), Ndof, 1);

```

7.3.3 双线性形式第一项的计算

按照上面说的排列方式，单元分析与 Morley 元完全相同。此时

$$\varphi_{3+i} = \xi_i, \quad \varphi_{6+i} = \zeta_i, \quad i = 1, 2, 3.$$

单元刚度矩阵为

$$K_\beta = \int_\beta B^T R B dxdy = (k_{ij})_{9 \times 9},$$

其中,

$$k_{ij} = D \int_{\beta} \partial_{11}\varphi_i \partial_{11}\varphi_j + \partial_{22}\varphi_i \partial_{22}\varphi_j + \nu(\partial_{11}\varphi_i \partial_{22}\varphi_j + \partial_{22}\varphi_i \partial_{11}\varphi_j) + 2(1-\nu)\partial_{12}\varphi_i \partial_{12}\varphi_j.$$

与 Morley 元不同的是, 此时被积函数不是常数. 每个二阶导数都形如 $f(\lambda_1, \lambda_2, \lambda_3)$, 可用三角形上的 Gauss 求积计算.

为了计算基函数的二阶导, 从前面基函数的表达式可以看到, 这归结于计算三次齐次式的二阶导

$$\partial_{11}(\lambda_i \lambda_j \lambda_k), \quad \partial_{22}(\lambda_i \lambda_j \lambda_k), \quad \partial_{12}(\lambda_i \lambda_j \lambda_k), \quad 1 \leq i, j, k \leq 3$$

以及 λ_i^2 的二阶导.

1. 我们考虑一般情形的二次齐次式 $\lambda_i \lambda_j$. 直接计算可知, $\partial_{\alpha\beta}(\lambda_i \lambda_j)$ 由两部分组成

- 对同一个求二阶导

$$(\partial_{\alpha\beta} \lambda_i) \lambda_j + \lambda_i (\partial_{\alpha\beta} \lambda_j);$$

- 对不同位置求导

$$(\partial_\beta \lambda_i)(\partial_\alpha \lambda_j) + (\partial_\alpha \lambda_i)(\partial_\beta \lambda_j).$$

注意到 λ_i 是一次多项式, 第一部分为零, 故

$$\partial_{\alpha\beta}(\lambda_i \lambda_j) = (\partial_\beta \lambda_i)(\partial_\alpha \lambda_j) + (\partial_\alpha \lambda_i)(\partial_\beta \lambda_j).$$

归结为 λ_i 的一阶导的计算.

2. 再考虑三次齐次式. $\partial_{\alpha\beta}(\lambda_i \lambda_j \lambda_k)$ 的和项由如下部分组成

- 对同一个求二阶导

$$(\partial_{\alpha\beta} \lambda_i) \lambda_j \lambda_k + \lambda_i (\partial_{\alpha\beta} \lambda_j) \lambda_k + \lambda_i \lambda_j (\partial_{\alpha\beta} \lambda_k);$$

- 对 1,2 位置求导

$$(\partial_\alpha \lambda_i)(\partial_\beta \lambda_j) \lambda_k + (\partial_\beta \lambda_i)(\partial_\alpha \lambda_j) \lambda_k;$$

- 对 2,3 位置求导

$$\lambda_i (\partial_\alpha \lambda_j)(\partial_\beta \lambda_k) + \lambda_i (\partial_\beta \lambda_j)(\partial_\alpha \lambda_k);$$

- 对 3,1 位置求导

$$(\partial_\alpha \lambda_i) \lambda_j (\partial_\beta \lambda_k) + (\partial_\beta \lambda_i) \lambda_j (\partial_\alpha \lambda_k).$$

类似地, 第一部分为零. 这样, 它们的二阶导又归结为 λ_i 的一阶导的计算. 易知有

$$\partial_1 \lambda_i = \frac{\eta_i}{2S}, \quad \partial_2 \lambda_i = -\frac{\xi_i}{2S}, \quad i = 1, 2, 3,$$

它们都是常数.

先计算面积坐标函数的一阶导. 这在前面章节的讨论中已经给出了, 函数为 gradbasis.m (节 5.2.1). 它给出所有单元的结果且以三维数组存储 (事实上还输出了面积), 形式如下

```

1 Dlambda(1:NT,:,:,1) = grad1;
2 Dlambda(1:NT,:,:,2) = grad2;
3 Dlambda(1:NT,:,:,3) = grad3;
```

它的 3 个部分都是 $NT \times 2$ 的矩阵, 该矩阵的第一列存储 ∂_1 的结果, 第二列则为 ∂_2 的.

接着计算

$$\partial_{\alpha\beta}(\lambda_i \lambda_j) = (\partial_\beta \lambda_i)(\partial_\alpha \lambda_j) + (\partial_\alpha \lambda_i)(\partial_\beta \lambda_j),$$

它显然为常数. 为了方便, 我们用函数 quadbasis.m 计算 (i, j) 配对的所有导数 (所有单元的结果), 如下排列

$$[\partial_{11}(\lambda_i \lambda_j), \partial_{22}(\lambda_i \lambda_j), \partial_{12}(\lambda_i \lambda_j)],$$

这里每列都是所有单元的. 考虑到 Gauss 积分点的问题, 该结果复制 nQuad 列, 每列对应一个积分点. 我们用三维数组存储, 维数为 $NT \times nQuad \times 3$

```

1 function Dquad = quadbasis(i,j,Dlambda,lambda)
2 % second derivatives of lambda_i*lambda_j
3 NT = size(Dlambda,1); nI = size(lambda,1);
4 Dquad = zeros(NT,nI,3); % [11,22,12]
5 Dquad(1:NT,:,:,1) = repmat(2*Dlambda(:,1,i).*Dlambda(:,1,j),1,nI);
6 Dquad(1:NT,:,:,2) = repmat(2*Dlambda(:,2,i).*Dlambda(:,2,j),1,nI);
7 Dquad(1:NT,:,:,3) = repmat((Dlambda(:,1,i).*Dlambda(:,2,j) +
8 Dlambda(:,2,i).*Dlambda(:,1,j)), 1,nI);
8 end
```

还要准备所有三次齐次式的二阶导在 Gauss 积分点处的值. 为了方便, 我们定义一个函数 tribasis.m. 它的输入是齐次式的下标 (i, j, k) , 返回的是三个二阶导在积分点处的值 (所有单元). 同样用三维数组存储, 维数为 $NT \times nQuad \times 3$, 其中, $nQuad$ 是积分点个数, 3 对应三个二阶导 $[\partial_{11}, \partial_{22}, \partial_{12}]$.

```

1 function Dtri = tribasis(i,j,k,Dlambda,lambda)
2 % second derivatives of lambda_i*lambda_j*lambda_k
3 NT = size(Dlambda,1); nI = size(lambda,1);
4 Dtri = zeros(NT,nI,3); % [11,22,12]
5 ss = [1 2 1]; tt = [1 2 2];
```

```

6   for m = 1:3 % loop for [11,22,12]
7     s = ss(m); t = tt(m);
8     b1 = ...
9       (Dlambda(:,s,i).*Dlambda(:,t,j)+Dlambda(:,t,i).*Dlambda(:,s,j))*lambda(:,k)';
10    b2 = ...
11      (Dlambda(:,s,j).*Dlambda(:,t,k)+Dlambda(:,t,j).*Dlambda(:,s,k))*lambda(:,i)';
12    b3 = ...
13      (Dlambda(:,s,i).*Dlambda(:,t,k)+Dlambda(:,t,i).*Dlambda(:,s,k))*lambda(:,j)';
14    Dtri(1:NT,:,m) = b1 + b2 + b3;
15
16 end
17
18 end

```

上面两个函数可进一步写成匿名函数, 以方便输入不同配对, 如下

```

1 Dquad = @(i,j) quadbasis(i,j,Dphi,lambda);
2 Dtri = @(i,j,k) tribasis(i,j,k,Dphi,lambda);

```

这样, 基函数的二阶导数可如下计算

```

1 %% Second derivatives of basis functions
2 b11 = zeros(NT,Ndof,nQuad); b22 = b11; b12 = b11;
3 for i = 1:3
4   % \phi (11,22,12)
5   DD = 3*Dquad(i,i)-2*Dtri(i,i,i)+2*Dtri(1,2,3);
6   b11(:,i,:) = DD(:, :, 1);
7   b22(:,i,:) = DD(:, :, 2);
8   b12(:,i,:) = DD(:, :, 3);
9   % \psi (11,22,12)
10  cc1(:,:,1) = repmat(c1,1,nQuad); cc1(:,:,2) = cc1(:,:,1); cc1(:,:,3) = ...
11    cc1(:,:,1);
12  cc2(:,:,1) = repmat(c2,1,nQuad); cc2(:,:,2) = cc2(:,:,1); cc2(:,:,3) = ...
13    cc2(:,:,1);
14  cc3(:,:,1) = repmat(c3(:,i),1,nQuad); cc3(:,:,2) = cc3(:,:,1); cc3(:,:,3) = ...
15    cc3(:,:,1);
16  cc4(:,:,1) = repmat(c4(:,i),1,nQuad); cc4(:,:,2) = cc4(:,:,1); cc4(:,:,3) = ...
17    cc4(:,:,1);
18  DD = cc1.*Dtri(i,i,2)+cc2.*Dtri(i,i,1)+cc3.*Dquad(i,i)+cc4.*Dtri(1,2,3);
19  b11(:,3+i,:)=DD(:, :, 1);
20  b22(:,3+i,:)=DD(:, :, 2);
21  b12(:,3+i,:)=DD(:, :, 3);
22  % \zeta (11,22,12)
23  dd1(:,:,1) = repmat(d1,1,nQuad); dd1(:,:,2) = dd1(:,:,1); dd1(:,:,3) = ...
24    dd1(:,:,1);
25  dd2(:,:,1) = repmat(d2,1,nQuad); dd2(:,:,2) = dd2(:,:,1); dd2(:,:,3) = ...
26    dd2(:,:,1);
27  dd3(:,:,1) = repmat(d3(:,i),1,nQuad); dd3(:,:,2) = dd3(:,:,1); dd3(:,:,3) = ...
28    dd3(:,:,1);
29  dd4(:,:,1) = repmat(d4(:,i),1,nQuad); dd4(:,:,2) = dd4(:,:,1); dd4(:,:,3) = ...
30    dd4(:,:,1);
31  DD = dd1.*Dtri(i,i,2)+dd2.*Dtri(i,i,1)+dd3.*Dquad(i,i)+dd4.*Dtri(1,2,3);

```

```

24     b11(:,6+i,:) = DD(:,:,1);
25     b22(:,6+i,:) = DD(:,:,2);
26     b12(:,6+i,:) = DD(:,:,3);
27 end

```

有了基函数的二阶导，就可如下用 Gauss 积分计算积分

```

1 %% First stiffness matrix
2 K = zeros(NT,Ndof^2); s = 1;
3 for i = 1:Ndof
4     for j = 1:Ndof
5         for p = 1:nQuad
6             Kp = b11(:,i,p).*b11(:,j,p) + b22(:,i,p).*b22(:,j,p) ...
7                 + nu*(b11(:,i,p).*b22(:,j,p) + b22(:,i,p).*b11(:,j,p)) ...
8                 + 2*(1-nu)*b12(:,i,p).*b12(:,j,p);
9             K(:,s) = K(:,s) + weight(p)*Kp;
10        end
11        s = s+1;
12    end
13 end
14 K = repmat(D*area,1,Ndof^2).*K;

```

7.3.4 双线性形式第二项的计算

类似 Morley 元可如下给出计算.

```

1 %% Second stiffness matrix and load vector
2 G = zeros(NT,Ndof^2); F = zeros(NT,Ndof);
3 if isnumeric(para.c), cf = @ (xy) para.c+0*xy(:,1); end
4 for p = 1:nQuad
5     % quadrature points in the x-y coordinate
6     pxy = lambda(p,1)*node(elem(:,1),:) ...
7         + lambda(p,2)*node(elem(:,2),:) ...
8         + lambda(p,3)*node(elem(:,3),:);
9     % basis functions at the p-th quadrature point
10    base = zeros(NT,Ndof);
11    for i = 1:3
12        % \phi
13        lam123 = lambda(p,1).*lambda(p,2).*lambda(p,3);
14        base(:,i) = lambda(p,i).^2.* (3-2*lambda(p,i)) + 2*lam123;
15        % \psi
16        base(:,3+i) = lambda(p,i).^2.* (c1*lambda(p,2) + c2*lambda(p,1) + c3(:,i)) ...
17                    + c4(:,i)*lam123;
18        % \zeta
19        base(:,6+i) = lambda(p,i).^2.* (d1*lambda(p,2) + d2*lambda(p,1) + d3(:,i)) ...
20                    + d4(:,i)*lam123;
21    end
22    % Second stiffness matrix
23    s = 1;

```

```

24     for i = 1:Ndof
25         for j = 1:Ndof
26             gs = cf(pxy).*base(:,i).*base(:,j);
27             G(:,s) = G(:,s) + weight(p)*gs;
28         end
29         s = s+1;
30     end
31 % load vector
32 F = F + weight(p)*repmat(f(pxy),1,Ndof).*base;
33 end
34 G = repmat(area,1,Ndof^2).*G;
35 F = repmat(area,1,Ndof).*F;

```

这里, c_i 和 d_i 是基函数的系数, 如下

```

1 % coefficients in the basis functions
2 c1 = xi(:,1); d1 = eta(:,1);
3 c2 = -xi(:,2); d2 = -eta(:,2);
4 c3 = -[c2,c1,zeros(NT,1)]; d3 = -[d2,d1,zeros(NT,1)];
5 c4 = [0.5*c1-c2, 0.5*c2-c1, 0.5*(c1+c2)];
6 d4 = [0.5*d1-d2, 0.5*d2-d1, 0.5*(d1+d2)];

```

7.3.5 边界条件的处理

假设直接给出边界上对应自由度的值, 则按照 Dirichlet 边界条件处理即可.

```

1 %% Dirichlet boundary conditions
2 bdNodeIdx = bdStruct.bdNodeIdx;
3 id = [bdNodeIdx; bdNodeIdx+N; bdNodeIdx+2*N];
4 isBdDof = false(3*N,1); isBdDof(id) = true;
5 bdDof = isBdDof; freeDof = (~isBdDof);
6 g_D = pde.g_D; pD = node(bdNodeIdx,:); Dw = pde.Dw;
7 wD = g_D(pD); Dw = Dw(pD); %wxD = Dw(:,1); wyD = Dw(:,2);
8 w = zeros(3*N,1); w(bdDof) = [wD; Dw(:)];
9 ff = ff - kk*w;
10
11 %% Solver
12 w(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

数值结果与程序不再给出, 主程序为 main_PlateBendingZienkiewicz.m, 函数文件为 PlateBendingZienkiewicz.m.

7.4 非协调 Adini 元

Adini 元是不完全双三次矩形元, 它是 C^0 连续的. 工程中通常称为 Adini-Clough-Melosh 元, 简称 ACM 元. 本文称为 Adini 元.

7.4.1 Adini 元的构造

设矩阵 $\beta = (\delta_1, \delta_2, \delta_3, \delta_4)$, $\delta_i = (x_i, y_i)$, 它的长宽平行于坐标轴. 自由度为

$$w(\delta_i) = w_i, \quad \frac{\partial w}{\partial x}(\delta_i) = w_{xi}, \quad \frac{\partial w}{\partial y}(\delta_i) = w_{yi}, \quad i = 1, 2, 3, 4.$$

完全三次多项式有 16 个自由度, 去掉其中 4 个高阶项

$$x^3y^3, \quad x^3y^2, \quad x^2y^3, \quad x^2y^2,$$

即可构造 12 个自由度的多项式. 显然形函数空间可写为

$$\mathcal{P} = \mathbb{P}_2 \oplus \{x^3y, xy^3\}.$$

做坐标变换

$$\begin{cases} \xi = \frac{2}{a}(x - x_0), & x_0 = \frac{1}{2}(x_1 + x_2), & a = x_2 - x_1, \\ \eta = \frac{2}{b}(y - y_0), & y_0 = \frac{1}{2}(y_1 + y_4), & b = y_4 - y_1, \end{cases}$$

矩形 β 变为参考元 $[-1, 1]^2$, 相应的四个顶点为

$$(\xi_1, \eta_1) = (-1, -1), \quad (\xi_2, \eta_2) = (1, -1), \quad (\xi_3, \eta_3) = (1, 1), \quad (\xi_4, \eta_4) = (-1, 1).$$

在 ξ, η 坐标下, 基函数为

$$\begin{cases} \varphi_i(\xi, \eta) = \frac{1}{8} (1 + \xi_i \xi) (1 + \eta_i \eta) (2 + \xi_i \xi + \eta_i \eta - \xi^2 - \eta^2), \\ \psi_i(\xi, \eta) = -\frac{1}{16} a \xi_i (1 + \xi_i \xi) (1 + \eta_i \eta) (1 - \xi^2), \\ \zeta_i(\xi, \eta) = -\frac{1}{16} b \eta_i (1 + \xi_i \xi) (1 + \eta_i \eta) (1 - \eta^2). \end{cases}$$

简单计算知, 二阶导数为

$$\begin{cases} \partial_{11}\varphi_i = -\frac{3}{a^2} \xi_i \xi (1 + \eta_i \eta) \\ \partial_{22}\varphi_i = -\frac{3}{b^2} \eta_i \eta (1 + \xi_i \xi) \quad , \quad i = 1, 2, 3, 4, \\ \partial_{12}\varphi_i = \frac{1}{2ab} \xi_i \eta_i [4 - 3(\xi^2 + \eta^2)] \\ \partial_{11}\psi_i = \frac{1}{2a} (1 + \eta_i \eta) (\xi_i + 3\xi) \\ \partial_{22}\psi_i = 0 \quad , \quad i = 1, 2, 3, 4, \\ \partial_{12}\psi_i = -\frac{1}{4b} \eta_i (1 - 2\xi_i \xi - 3\xi^2) \\ \partial_{11}\zeta_i = 0 \\ \partial_{22}\zeta_i = \frac{1}{2b} (1 + \xi_i \xi) (\eta_i + 3\eta) \quad , \quad i = 1, 2, 3, 4. \\ \partial_{12}\zeta_i = -\frac{1}{4a} \xi_i (1 - 2\eta_i \eta - 3\eta^2) \end{cases}$$

注 7.4 Adini 元与 Zienkiewicz 元的自由度形式一样, `sparse` 装配指标完全相同.

7.4.2 刚度矩阵和载荷向量的计算

与三角形不同的是, 现在是转换到参考矩形 $[-1, 1]^2$ 上, 易知

$$\int_{\beta} f(x, y) dx dy = \int_{-1}^1 \int_{-1}^1 \tilde{f}(\xi, \eta) |J| d\xi d\eta, \quad |J| = \frac{ab}{4}.$$

设 $[-1, 1]$ 上的 Gauss 点和权重为 $r_i, w_i, i = 1, \dots, n$, 则二重积分如下计算

$$\int_{-1}^1 \int_{-1}^1 f(\xi, \eta) d\xi d\eta = \sum_{i,j=1}^n w_i w_j f(r_i, r_j).$$

具体计算时, 权重 $\{w_i w_j\}$ 将拉直为一个向量, 记为 `weight`, 相应的积分点的横纵坐标分别记为 `xi` 和 `eta`. 如下实现

```

1 % quadrature points on all elements [xa, xb]
2 [lambda, weight] = quadpts1(5);
3 xx = (2*lambda(:,1)-1)'; % each row corresponds to an element
4 ww = weight;
5 nr = length(ww); nQuad = nr^2;
6 xi = reshape(repmat(xx',nr,1),1,[]);
7 eta = reshape(ones(nr,1)*xx,1,[]);
8 weight = reshape(ww'*ww,1,[]);

```

所有矩形的边长为

```

1 % area
2 xiv = [-1 1 1 -1]; etav = [-1 -1 1 1]; % vertices of [-1,1]^2
3 x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
4 x2 = node(elem(:,2),1); y4 = node(elem(:,4),2);
5 a = x2-x1; b = y4-y1;
6 area = a.*b;

```

基函数的二阶导数类似 Zienkiewicz 元存储为三维数组, 且如下计算

```

1 %% second derivatives of basis functions
2 b11 = zeros(NT,Ndof,nQuad); b22 = b11; b12 = b11;
3 for i = 1:4
4     % \phi
5     b11(:,i,:) = -3./a.^2*(xiv(i)*xi.*(1+etav(i)*eta));
6     b22(:,i,:) = -3./b.^2*(etav(i)*eta.*(1+xiv(i)*xi));
7     b12(:,i,:) = 1./(2*a.*b)*(xiv(i)*etav(i)*(4-3*(xi.^2+eta.^2)));
8     % \psi
9     b11(:,4+i,:) = 1./(2*a)*((1+etav(i)*eta).*(xiv(i)+3*xi));
10    b22(:,4+i,:) = 0;
11    b12(:,4+i,:) = -1./(4*b)*(etav(i)*(1-2*xiv(i)*xi-3*xi.^2));
12    % \zeta
13    b11(:,8+i,:) = 0;
14    b22(:,8+i,:) = 1./(2*b)*((1+xiv(i)*xi).*(etav(i)+3*eta));
15    b12(:,8+i,:) = -1./(4*a)*(xiv(i)*(1-2*etav(i)*eta-3*eta.^2));
16 end

```

双线性形式的第一部分也类似 Zienkiewicz 计算, 只不过积分点换成矩形的而已.

```
1 %% First stiffness matrix
2 K = zeros(NT,Ndof^2);
3 s = 1;
4 for i = 1:Ndof
5     for j = 1:Ndof
6         for p = 1:nQuad
7             Kp = b11(:,i,p).*b11(:,j,p) + b22(:,i,p).*b22(:,j,p) ...
8                 + nu*(b11(:,i,p).*b22(:,j,p) + b22(:,i,p).*b11(:,j,p)) ...
9                 + 2*(1-nu)*b12(:,i,p).*b12(:,j,p);
10            K(:,s) = K(:,s) + weight(p)*Kp;
11        end
12        s = s+1;
13    end
14 end
15 K = repmat(D*(area)./4,1,Ndof^2).*K;
```

第二部分与载荷向量如下计算

```
1 %% Second stiffness matrix and load vector
2 G = zeros(NT,Ndof^2); F = zeros(NT,Ndof);
3 if isnumeric(para.c), cf = @ (xy) para.c+0*xy(:,1); end
4 x0 = (x1+x2)./2; y0 = (y1+y4)./2;
5 for p = 1:nQuad
6     % quadrature points in the x-y coordinate
7     x = a*xi(p)/2 + x0; y = b*eta(p)/2 + y0;
8     pxy = [x,y];
9     % basis functions at the p-th quadrature point
10    base = zeros(NT,Ndof);
11    for i = 1:4
12        tp = (1+xiv(i)*xi(p))*(1+etav(i)*eta(p));
13        % \phi
14        base(:,i) = 1/8*tp*(2+xiv(i)*xi(p)+etav(i)*eta(p)-xi(p)^2-eta(p)^2);
15        % \psi
16        base(:,4+i) = -1/16*a*xiv(i)*tp*(1-xi(p)^2);
17        % \zeta
18        base(:,8+i) = -1/16*b*etav(i)*tp*(1-eta(p)^2);
19    end
20    % Second stiffness matrix
21    s = 1;
22    for i = 1:Ndof
23        for j = 1:Ndof
24            gs = cf(pxy).*base(:,i).*base(:,j);
25            G(:,s) = G(:,s) + weight(p)*gs;
26            s = s+1;
27        end
28    end
29    % load vector
30    F = F + weight(p)*repmat(f(pxy),1,Ndof).*base;
```

```

31 end
32 G = repmat(area./4,1,Ndof^2).*G;
33 F = repmat(area./4,1,Ndof).*F;

```

装配和边界条件处理与 Zienkiewicz 元完全相同. 数值结果与程序不再给出, 见 GitHub 上传文件. 主程序为 main_PlateBendingAdini.m, 函数文件为 PlateBendingAdini.m.

7.5 双调和方程的混合元方法

7.5.1 混合元的变分问题

考虑问题

$$\begin{cases} \Delta^2 u = f & \text{in } \Omega \subset \mathbb{R}^2, \\ u = \partial_n u = 0 & \text{on } \partial\Omega. \end{cases}$$

令 $w = -\Delta u$ (通常称 u 为流函数, w 为涡函数), 则

$$\begin{cases} -\Delta u = w, \\ -\Delta w = f, \\ u = \partial_n u = 0 & \text{on } \partial\Omega, \end{cases}$$

于是有混合变分问题: 找 $(w, u) \in H^1(\Omega) \times H_0^1(\Omega)$ 使得

$$\begin{cases} \int_{\Omega} \nabla u \cdot \nabla \phi dx = \int_{\Omega} w \phi dx, & \phi \in H^1(\Omega), \\ \int_{\Omega} \nabla w \cdot \nabla \psi dx = \int_{\Omega} f \psi dx, & \psi \in H_0^1(\Omega). \end{cases} \quad (7.7)$$

令

$$a(w, \phi) = - \int_{\Omega} w \phi dx, \quad b(\phi, u) = \int_{\Omega} \nabla \phi \cdot \nabla u dx,$$

则

$$\begin{cases} a(w, \phi) + b(\phi, u) = 0, & \phi \in H^1(\Omega), \\ b(w, \psi) = (f, \psi), & \psi \in H_0^1(\Omega). \end{cases}$$

注意, 若令 $V = H^1(\Omega)$, $U = H_0^1(\Omega)$, 则

$$a(\cdot, \cdot) : V \times V \rightarrow \mathbb{R}, \quad b(\cdot, \cdot) : V \times U \rightarrow \mathbb{R}.$$

对 u, w , 以下均采用线性元近似. 现在考察其刚度矩阵和载荷向量. 设 N 是单元上节点基函数的行向量, 则有

$$\begin{cases} \phi^T A w + \phi^T B u = 0, \\ \psi^T B^T w = \psi^T f, \end{cases}$$

式中,

$$A = - \int_K N^T N dx, \quad B = \int_K \nabla N^T \cdot \nabla N dx, \quad \mathbf{f} = \int_K N^T f dx.$$

此即,

$$\begin{bmatrix} A & B \\ B^T & O \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{f} \end{bmatrix}.$$

总体方程组也是该形式.

注 7.5 若 $\partial_n u$ 不为零, 则混合变分形式为

$$\begin{cases} a(w, \phi) + b(\phi, u) = \int_{\partial\Omega} \partial_n u \phi ds, & \phi \in H^1(\Omega), \\ b(w, \psi) = (f, \psi), & \psi \in H_0^1(\Omega). \end{cases}$$

现在 $\partial_n u$ 成为 u 对应的 Neumann 边界条件, 贡献在第一行方程中.

7.5.2 装配指标

类似线弹性问题, 我们分块进行装配, 以方便运用向量运算. 装配指标为

```

1 %% Sparse assembling index
2 ii = reshape(repmat(elem, Ndof, 1), [], 1);
3 jj = repmat(elem(:, 1));
4 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
5 ii21 = ii+N; jj21 = jj;

```

这里第 4 块为零, 不需要给出.

载荷向量的第二部分为零, 只需要如下装配

```

1 ff = accumarray(elem(:)+N, F(:, [2*N 1]));

```

7.5.3 刚度矩阵和载荷向量的计算

矩阵 A , B 等的计算都是标准的, 直接给出.

```

1 %% Stiffness matrix B
2 % B: (Dphi_i, Dphi_j)_K
3 [Dphi, area] = gradbasis(node, elem);
4 B = zeros(NT, Ndof^2); s = 1;
5 for i = 1:Ndof
6     for j = 1:Ndof
7         B(:, s) = sum(Dphi(:, :, i).*Dphi(:, :, j), 2).*area;
8         s = s+1;
9     end
10 end
11

```

```

12 %% Stiffness matrix A and load vector
13 % Gauss quadrature rule
14 [lambda,weight] = quadpts(3); nG = length(weight);
15 % A: -(phi_i, phi_j)_K
16 A = zeros(NT,Ndof^2); F = zeros(NT,Ndof);
17 for p = 1:nG
18     % quadrature points in the x-y coordinate
19     pxy = lambda(p,1)*node(elem(:,1),:) ...
20         + lambda(p,2)*node(elem(:,2),:) ...
21         + lambda(p,3)*node(elem(:,3),:);
22     % Second stiffness matrix
23     s = 1;
24     for i = 1:Ndof
25         for j = 1:Ndof
26             gs = lambda(p,i).*lambda(p,j);
27             A(:,s) = A(:,s) + weight(p)*gs;
28             s = s+1;
29         end
30     end
31     % load vector
32     F = F + weight(p)*pde.f(pxy)*lambda(p,:);
33 end
34 A = -repmat(area,1,Ndof^2).*A;
35 F = repmat(area,1,Ndof).*F;
36
37 %% Assemble stiffness matrix and load vector
38 ss11 = A(:,1); ss12 = B(:,1); B1 = B(:,[1 4 7 2 5 8 3 6 9]);
39 ss21 = B1(:,1);
40 ii = [ii11; ii12; ii21];
41 jj = [jj11; jj12; jj21];
42 ss = [ss11; ss12; ss21];
43 kk = sparse(ii,jj,ss,2*N,2*N);
44 ff = accumarray(elem(:)+N, F(:,[2*N 1]));

```

7.5.4 边界条件的处理

Neumann 边界条件位于第一行方程, 如下处理.

```

1 %% Assemble Neumann boundary conditions
2 EdgeN = bdStruct.bdEdgeD;
3 z1 = node(EdgeN(:,1),:); z2 = node(EdgeN(:,2),:);
4 e = z1-z2; % e = z2-z1
5 ne = [-e(:,2),e(:,1)]; % scaled ne
6 Du = pde.Du;
7 gradu1 = Du(z1); gradu2 = Du(z2);
8 F1 = sum(ne.*gradu1,2)./2; F2 = sum(ne.*gradu2,2)./2;
9 FN = [F1,F2];
10 ff = ff + accumarray(EdgeN(:), FN(:,[2*N 1]));

```

Dirichlet 边界条件比较容易.

```
1 %% Apply Dirichlet boundary conditions
2 bdNodeIdx = bdStruct.bdNodeIdx; g_D = pde.g_D;
3 id = bdNodeIdx+N;
4 isBdDof = false(2*N,1); isBdDof(id) = true;
5 bdDof = isBdDof; freeDof = (~isBdDof);
6 pD = node(bdNodeIdx,:);
7 U = zeros(2*N,1); U(bdDof) = g_D(pD);
8 ff = ff - kk*U;
9
10 %% Set solver
11 U(freeDof) = kk(freeDof,freeDof)\ff(freeDof);
12 u = U(N+1:end); w = U(1:N);
```

7.5.5 程序整理

函数文件为 biharmonicMixedFEM.m, 这里不再给出. 误差阶如下计算.

```
1 clc;clear;close all;
2 % ----- Mesh -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 Nx = 4; Ny = 4; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
6
7 % ----- PDE data -----
8 pde = biharmonicdata;
9
10 % ----- biharmonicMixedFEM -----
11 maxIt = 5;
12 N = zeros(maxIt,1); h = zeros(maxIt,1);
13 erruL2 = zeros(maxIt,1); erruH1 = zeros(maxIt,1);
14 errwL2 = zeros(maxIt,1); errwH1 = zeros(maxIt,1);
15 for k = 1:maxIt
16     [node,elem] = uniformrefine(node,elem);
17     bdStruct = setboundary(node,elem);
18     [u,w] = biharmonicMixedFEM(node,elem,pde,bdStruct);
19     NT = size(elem,1);
20     h(k) = 1/sqrt(NT);
21     erruL2(k) = getL2error(node,elem,u,pde.uexact);
22     erruH1(k) = getH1error(node,elem,u,pde.Du);
23     errwL2(k) = getL2error(node,elem,w,pde.wexact);
24     errwH1(k) = getH1error(node,elem,w,pde.Dw);
25 end
26
27 % ----- Plot convergence rates -----
28 figure;
29 subplot(1,2,1);
30 showrateh(h, erruL2, erruH1, '|| u - u_h ||', '|| Du - Du_h ||');
31 subplot(1,2,2)
```

```
32 showrateh(h, errwL2, errwH1, '||| w - w_h |||', '||| Dw - Dw_h |||');
```

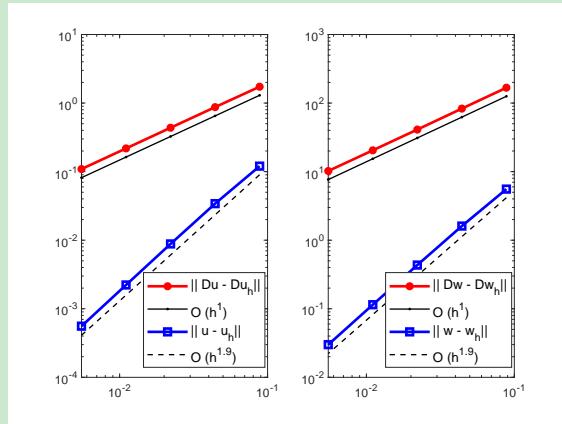


图 7.3. 双调和方程一阶混合元方法的收敛性

从上图可以看到, 一阶混合元方法对 u 和 w 来说, 误差阶都是最优的 (后面将看到, u 对 P1,P2,P3-Lagrange 元都是最优的, 但对 w 是次优的).

Part III

基于变分形式的有限元程序设计

第八章 基于变分形式的有限元程序设计

FreeFEM 是一款非常优秀的有限元分析软件, 它的一大亮点是编程过程与变分形式一一对应, 本文称为“基于变分形式的编程”. 这种处理方式通常使用面向对象的语言, 程序组织以及阅读起来都非常困难. 对有限元编程过程的再思考之后, 笔者觉得也可以用面向过程的语言写出基于变分形式的程序, 其本质在于处理各种典型的双线性形式以及线性形式相对于基函数的积分.

8.1 程序的设计思路

8.1.1 一些重要观察

程序设计基于几个重要观察。

观察一: 检验函数的前后问题

设

$$v = \sum_{i=1}^m v_i \varphi_i, \quad u = \sum_{i=1}^m u_i \phi_i,$$

则

$$a(v, u) = \mathbf{v}^T \mathbf{A} \mathbf{u}, \quad \mathbf{A} = (a(\varphi_i, \phi_j))_{m \times n}.$$

可以看到, 从计算角度来说, 把检验函数 v 写在配对的前面是最自然的. 为此, 以下将变分形式的检验函数写在前面, 试探函数写在后面.

观察二: 双线性形式计算的归结

考虑几个典型问题的变分或双线性形式:

- Poisson 方程

$$a(v, u) = \int_{\Omega} a \nabla u \cdot \nabla v d\sigma + \int_{\Omega} c u v d\sigma + \int_{\Gamma_R} g_R u v ds.$$

- 线弹性边值问题

$$a(\mathbf{v}, \mathbf{u}) = 2\mu \int_{\Omega} \varepsilon_{ij}(\mathbf{v}) \varepsilon_{ij}(\mathbf{u}) dx + \lambda \int_{\Omega} (\partial_i v_i)(\partial_j u_j) dx,$$

其中,

$$\int_{\Omega} \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx.$$

- 双调和方程混合元方法

$$\begin{cases} -\int_{\Omega} w \phi dx + \int_{\Omega} \nabla u \cdot \nabla \phi dx = \int_{\partial\Omega} \partial_n u \phi ds, & \phi \in H^1(\Omega), \\ \int_{\Omega} \nabla w \cdot \nabla \psi dx = \int_{\Omega} f \psi dx, & \psi \in H_0^1(\Omega). \end{cases}$$

- Stokes 方程

$$\begin{aligned} & a_1(v_1, u_1) + a_2(v_2, u_2) \\ & + b_1(v_1, p) + b_2(v_2, p) \\ & + b_1(q, u_1) + b_2(q, u_2) \\ & - \varepsilon(q, p) \\ & = F_1(v_1) + F_2(v_2), \end{aligned}$$

这里 a_i, b_i 等的具体定义略.

可以看到, 双线性形式的计算本质上归结为标量情形的某种双线性形式 $a(v, u)$ 的计算, 这里 v 和 u 容许位于不同有限元空间, 如 Stokes 问题的 $b_1(v_1, p)$. 线性形式的计算类似.

观察三: 双线性形式的装配

假设 $\mathbf{v} = (v_1, v_2, v_3)$, $\mathbf{u} = (u_1, u_2, u_3)$, $a(\mathbf{v}, \mathbf{u})$ 是一个双线性函数. 要注意的是, 一般而言, v_i ($i = 1, 2, 3$) 可以不在同一个空间, 但 v_i 与 u_i 是在同一个空间的, 否则刚度矩阵就不是方阵了.

刚度矩阵经分块后有如下对应

$$a(\mathbf{v}, \mathbf{u}) \leftrightarrow [\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \end{bmatrix},$$

其中 \mathbf{u}_i 是 u_i 在节点基下的自由度向量. 易知 A_{ij} 由 $a(\mathbf{v}, \mathbf{u})$ 中所有配对为 (v_i, u_j) 的双线性形式按标量情形装配所得.

设 A_{ij} 对应的标量情形的稀疏装配指标为 $(\mathbf{i}_{ij}, \mathbf{j}_{ij})$, $A_{i\cdot}$ 的行数为 m_i , $A_{\cdot j}$ 的列数为 n_j , 则最终的稀疏装配指标按块对应为

$$\begin{bmatrix} \mathbf{i}_{11} & \mathbf{i}_{12} & \mathbf{i}_{13} \\ \mathbf{i}_{21} + m_1 & \mathbf{i}_{22} + m_1 & \mathbf{i}_{23} + m_1 \\ \mathbf{i}_{31} + m_2 & \mathbf{i}_{32} + m_2 & \mathbf{i}_{33} + m_2 \end{bmatrix}, \quad \begin{bmatrix} \mathbf{j}_{11} & \mathbf{j}_{12} + n_1 & \mathbf{j}_{13} + n_2 \\ \mathbf{j}_{21} & \mathbf{j}_{22} + n_1 & \mathbf{j}_{23} + n_2 \\ \mathbf{j}_{31} & \mathbf{j}_{32} + n_1 & \mathbf{j}_{33} + n_2 \end{bmatrix},$$

它们要按行拉直进行拼接.

8.1.2 变分形式与程序的对应

考虑如下的一般问题

$$\begin{cases} -\nabla \cdot (a \nabla u) + cu = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ g_R u + a \partial_n u = g_N & \text{on } \Gamma_R, \end{cases}$$

设区域 $\Omega = (0, 1)^2$, 且右侧对应 Γ_R , 其他都为 Γ_D . 相关 pde 数据见 PoissonDataVar.m. 对齐次 Dirichlet 边界条件, 变分问题为: 找 $u \in V := H_0^1(\Omega)$ 使得

$$a(v, u) = \ell(v), \quad v \in V,$$

式中,

$$\begin{aligned} a(v, u) &= \int_{\Omega} a \nabla v \cdot \nabla u d\sigma + \int_{\Omega} c v u d\sigma + \int_{\Gamma_R} g_R v u ds, \\ \ell(v) &= \int_{\Omega} f v d\sigma + \int_{\Gamma_R} g_N v ds. \end{aligned}$$

先考虑双线性形式. 第一步是获得区域上双线性形式

$$\int_{\Omega} a \nabla v \cdot \nabla u d\sigma + \int_{\Omega} c v u d\sigma$$

对应的刚度矩阵, 程序模式如下

```

1 % Omega
2 Coef  = {pde.a, pde.c};
3 Test   = {'v.grad', 'v.val'};
4 Trial  = {'u.grad', 'u.val'};
5 kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);

```

这里, 我们设置了三元组 $(\text{Coef}, \text{Test}, \text{Trial})$, 分别对应变分形式中的系数、检验函数和试探函数. 注意, 双线性形式三元组的元素始终用大括号括起来, 即便只有一个元素的对应. int2d 计算二维区域上双线性形式对应的刚度矩阵, 即

$$A = (a_{ij}), \quad a_{ij} = a(\Phi_i, \Phi_j),$$

其中的 Φ_i 是有限元空间 V_h 的整体节点基. 双线性形式的积分, 如

$$\int_{\Omega} \nabla \Phi_i \cdot \nabla \Phi_j dx$$

将采用 Gauss 积分公式进行近似计算, quadOrder 即为 Gauss 积分的阶.

接着是实现边界上的双线性形式

$$\int_{\Gamma_R} g_R v u ds$$

对应的刚度矩阵, 程序模式如下

```

1 % Gamma_R
2 Coef = {pde.g_R};
3 Test = {'v.val'};
4 Trial = {'u.val'};
5 kk = kk + int1d(Th,Coef,Test,Trial,Vh,quadOrder);

```

这里, int1d 计算一维单形剖分上双线性形式对应的刚度矩阵. 当然它的矩阵维数与区域上一致.

再考虑右端的线性形式, 它也分为两部分. 第一步是实现区域上线性形式

$$\int_{\Omega} f v d\sigma$$

对应的载荷向量, 程序模式如下

```

1 % Omega
2 Coef = pde.f; Test = 'v.val';
3 ff = int2d(Th,Coef,[],Test,Vh,quadOrder);

```

线性形式一般都是上面的形式, 为此, 这里没有考虑多个组份, 从而 `v.val` 没有用元胞数组存储. 第二步是实现边界上线性形式

$$\int_{\Gamma_R} g_N v ds$$

对应的载荷向量, 程序模式如下

```

1 % Gamma_R
2 Coef = g_N; Test = 'v.val';
3 ff = ff + int1d(Th,Coef,[],[],Vh,quadOrder);

```

8.1.3 网格信息 Th

我们定义一个结构体 `Th`, 用以存储网格的相关信息, 对应的函数为 `getTh1D`, `getTh2D` 或 `getTh3D`.

1. 基本数据结构 `node` 和 `elem`.

考虑到二维几何的边界是一维几何, 三维几何的边界是二维几何, 我们规定:

- 结构体 `Th` 的域中, 一维单元、二维单元和三维单元分别用 `Th.elem1D`, `Th.elem2D` 和 `Th.elem3D` 表示.
- 但在各自的程序中, 我们仍使用 `elem` 表示, 例如一维程序中用 `elem = ... Th.elem1D 替换.`
- 高维问题在边界上用的仍是高维节点, 为此节点统一使用 `node` 表示.

2. 边界信息

- 我们总假设边界分为两部分, 即 Dirichlet 和 Neumann 边界, 这里 Neumann 可以是一般的 Robin 边界 (只是符号而已). 对 Dirichlet 边界条件, 我们要用到 Dirichlet 边界的顶点序号. 而且对高阶元, 例如二阶 Lagrange 元, 我们需要添加边中点自由度, 它对应的序号由边的序号获得. 为此, 我们实际上还需要给定 Dirichlet 边的序号. 对 Neumann 边界条件或边界积分, 我们需要 Neumann 边界的连通性信息. 当然对高阶元, 我们也要知道这些边的序号.
- 所有涉及到边的信息都存储在结构体 `bdStruct` 中, 对二维问题, 它包含如下信息:

```
1 bdStruct.bdEdge = bdEdge; % all boundary edges
2 bdStruct.elemD = bdEdge(bdFlag,:); % Dirichlet boundary edges
3 bdStruct.elemN = bdEdge(~bdFlag,:); % Neumann boundary edges
4 bdStruct.eD = unique(bdEdge(bdFlag,:)); % Dirichlet boundary nodes
5 bdIndex = find(s==1); % indices of all boundary edges
6 bdStruct.bdIndex = bdIndex;
7 bdStruct.bdIndexD = bdIndex(bdFlag); % indices of Dirichlet boundary edges
8 bdStruct.bdIndexN = bdIndex(~bdFlag); % indices of Neumann boundary edges
```

为此, `Th` 还新增第三个域: `Th.bdStruct = bdStruct`.

3. 辅助数据

对二维问题, 就目前而言, 我们只需要 `edge`, `elem2edge`, 它们存储在结构体 `auxT` 中.

一些常用的数字, 如 `N`, `NT`, `NE` 也保存在 `Th` 中, 如 `Th.NE` (一维问题没有).

注 8.1 网格的信息可能要根据具体情况添加信息. 我们建立了文件夹 `pre-post`, 用以存储一些可能需要前处理或后处理的程序; 或者新增有限元时, 需要添加内容的程序, 如 `Base2D.m`. 就目前而言, 它们不需要做调整.

8.2 assem2d 函数

本节将编写任意二维标量双线性形式

$$a(v, u), \quad v = \varphi_i, \quad u = \phi_j, \quad i = 1, \dots, m, \quad j = 1, \dots, n$$

的装配函数 `assem2d.m`, 这里的配对 v, u 容许对应不同有限元空间. 我们暂时只考虑最高三次的 Lagrange 元空间, 且把装配的程序命名为 `assem2d`. 对标量情形, `assem2d.m` 与 `int2d.m` 本质没有差别, 但为了适用向量情形, 我们额外设置该函数. 另外, 为了处理不同空间情形, 我们完全写出 (v, u) 形式的空间对, 如 `Vh = {'P1', 'P1'}`.

8.2.1 变分形式的计算

先讨论 v 和 u 位于相同空间的情形. 假设双线性形式仅含有一阶导, 可能的组合为

$$\begin{aligned} & \int_K avudx, \quad \int_K av_x u dx, \quad \int_K av_y u dx, \\ & \int_K avu_x dx, \quad \int_K av_x u_x dx, \quad \int_K av_y u_x dx, \\ & \int_K avu_y dx, \quad \int_K av_x u_y dx, \quad \int_K av_y u_y dx, \end{aligned}$$

当然我们经常用到梯度形式

$$\int_{\Omega} a \nabla v \cdot \nabla u dx = \int_{\Omega} a(v_x u_x + v_y u_y) dx.$$

以第二个双线性形式为例, 令

$$a_K(v, u) = \int_K av_x u dx.$$

考虑一阶 Lagrange 有限元, 设局部节点基为 ϕ_1, ϕ_2, ϕ_3 , 则单元矩阵为

$$[K^e] = \int_K a \begin{bmatrix} \partial_x \phi_1 \\ \partial_x \phi_2 \\ \partial_x \phi_3 \end{bmatrix} [\phi_1, \phi_2, \phi_3] dx.$$

记

$$v_1 = \partial_x \phi_1, \quad v_2 = \partial_x \phi_2, \quad v_3 = \partial_x \phi_3; \quad u_1 = \phi_1, \quad u_2 = \phi_2, \quad u_3 = \phi_3,$$

则

$$[K^e] = (k_{ij})_{3 \times 3}, \quad k_{ij} = \int_K av_i u_j dx.$$

根据数值积分的说明, 积分可如下计算

$$k_{ij} = \int_K av_i u_j dx = |K| \sum_{p=1}^{n_G} w_p a(x_p, y_p) v_i(x_p, y_p) u_j(x_p, y_p), \quad (8.1)$$

其中, (x_p, y_p) 是第 p 个 Gauss 积分点. 为了避免单元循环, 对固定的 p , 我们先一次性算出所有单元的第 p 个和项, 即

$$w_p \begin{bmatrix} v_i u_j|_{(x_p^1, y_p^1)} \\ v_i u_j|_{(x_p^2, y_p^2)} \\ \vdots \\ v_i u_j|_{(x_p^{NT}, y_p^{NT})} \end{bmatrix},$$

然后对 p 循环求和并将每行对应乘以单元面积即可. 事实上, 后面一次性生成所有积分节点处的. 为此, 我们可事先给出

$$w_p, \begin{bmatrix} v_i|_{(x_p^1, y_p^1)} \\ v_i|_{(x_p^2, y_p^2)} \\ \vdots \\ v_i|_{(x_p^{NT}, y_p^{NT})} \end{bmatrix}, \begin{bmatrix} u_j|_{(x_p^1, y_p^1)} \\ u_j|_{(x_p^2, y_p^2)} \\ \vdots \\ u_j|_{(x_p^{NT}, y_p^{NT})} \end{bmatrix}, \quad p = 1, \dots, n_G.$$

下面考虑 $v_i = \partial_x \phi_i$, $u_j = \phi_j$ 的计算程序. 设 i, j, k 表示 1,2,3 的轮换. 定义

$$\xi_i = x_j - x_k, \quad \eta_i = y_j - y_k, \quad \omega_i = x_j y_k - x_k y_j,$$

则三角形的有向面积可表示为

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} = \frac{1}{2} (\xi_1 \eta_2 - \xi_2 \eta_1) = \omega_1 \omega_2 \omega_3,$$

如下计算

```

1 z1 = node(elem(:,1),:); z2 = node(elem(:,2),:); z3 = node(elem(:,3),:);
2 xi = [z2(:,1)-z3(:,1), z3(:,1)-z1(:,1), z1(:,1)-z2(:,1)];
3 eta = [z2(:,2)-z3(:,2), z3(:,2)-z1(:,2), z1(:,2)-z2(:,2)];
4 area = 0.5*(xi(:,1).*eta(:,2)-xi(:,2).*eta(:,1));

```

基函数的表达式为

$$\phi_i(x, y) = \lambda_i(x, y) = \frac{1}{2S} (\eta_i x - \xi_i y + \omega_i), \quad i = 1, 2, 3,$$

显然有

$$\frac{\partial}{\partial x} \phi_i(x, y) = \frac{1}{2S} \eta_i, \quad \frac{\partial}{\partial y} \phi_i(x, y) = -\frac{1}{2S} \xi_i.$$

所有单元检验函数值如下

```

1 % test function
2 v1 = eta(:,1)./(2*area);
3 v2 = eta(:,2)./(2*area);
4 v3 = eta(:,3)./(2*area);

```

注意一阶导数是常数, 在所有积分点处的值是相同的, 为此我们直接复制 n_g 列.

```

1 % test function
2 v1 = -xi(:,1)./(2*area); v1 = repmat(v1, 1, ng);
3 v2 = -xi(:,2)./(2*area); v2 = repmat(v2, 1, ng);
4 v3 = -xi(:,3)./(2*area); v3 = repmat(v3, 1, ng);

```

这样, v_i 是一个矩阵, 每行对应一个单元, 每列对应一个积分点.

测试函数 $u_i = \phi_i$ 如下计算

```
1 % trial function
2 u1 = repmat(lambda(:,1)',NT,1);
3 u2 = repmat(lambda(:,2)',NT,1);
4 u3 = repmat(lambda(:,3)',NT,1);
```

注意, 这里 λ 的第 i 列对应 λ_i 的所有积分点值, 转置复制 NT 行就是所有单元的. 此时, u_j 每行对应一个单元, 每列对应一个积分点. 另外, iFEM 中权重 $weight$ 的维数未统一, 前面的若干个是行向量, 而后面的则是列向量, 这里统一修改为行向量.

先不考虑系数函数. 按照装配的规定, k_{ij} 如下排列

$$K = [k_{11}, k_{12}, k_{13}, k_{21}, k_{22}, k_{23}, k_{31}, k_{32}, k_{33}],$$

其中每个 k_{ij} 是一列向量, 对应所有单元的结果. 记 v_i 与 u_j 点乘后所得矩阵为 k_{ij} , 它的每行对应一个单元, 而每列恰好对应积分公式 (8.1) 右端的一个求和项, 只不过未乘以权重. 对应矩阵 k_{ij} , 我们可获得权重矩阵

```
1 ww = repmat(weight,NT,1);
```

注意权重 $weight$ 必须是一行向量, 列对应积分点. 复制 NT 行就是所有单元的结果. 这样, 将 ww 和 k_{ij} 点乘, 然后按行求和就是 (8.1) 右端的求和结果.

```
1 k11 = sum(ww.*v1.*u1,2);
2 k12 = sum(ww.*v1.*u2,2);
3 k13 = sum(ww.*v1.*u3,2);
4 k21 = sum(ww.*v2.*u1,2);
5 k22 = sum(ww.*v2.*u2,2);
6 k23 = sum(ww.*v2.*u3,2);
7 k31 = sum(ww.*v3.*u1,2);
8 k32 = sum(ww.*v3.*u2,2);
9 k33 = sum(ww.*v3.*u3,2);
10 K = [k11,k12,k13, k21,k22,k23, k31,k32,k33];
```

最后, 每行还要乘以面积

```
1 Ndof = 3;
2 K = repmat(area,1,Ndof^2).*K;
```

当然为了方便, 上面的过程可用一个简单的循环实现.

```
1 K = zeros(NT,Ndof^2); s = 1;
2 v = {v1,v2,v3}; u = {u1,u2,u3};
3 for i = 1:Ndof
4     for j = 1:Ndof
5         vi = v{i}; uj = u{j};
```

```

6      K(:,s) = area.*sum(ww.*vi.*uj,2);
7      s = s+1;
8  end
9 end

```

接着, 我们加上积分前的系数, 例如取 $a(x, y) = x + y$. 根据上面的说明, 只要获得类似 vi 或 uj 这样的系数矩阵即可, 它的每行对应一个单元, 每列对应一个积分点. 对第 p 个和项, 所有单元的积分点坐标 pz 如下

```

1 pz = lambda(p,1)*z1 + lambda(p,2)*z2 + lambda(p,3)*z3;

```

系数矩阵如下

```

1 cf = @(pz) pz(:,1) + pz(:,2); % x+y;
2 cc = zeros(NT,ng);
3 for p = 1:ng
4     pz = lambda(p,1)*z1 + lambda(p,2)*z2 + lambda(p,3)*z3;
5     cc(:,p) = cf(pz);
6 end

```

规定: 所有系数函数均采用向量型变量. 这样, 前面的循环可修改为

```

1 K = zeros(NT,Ndof^2); s = 1;
2 v = {v1,v2,v3}; u = {u1,u2,u3};
3 for i = 1:Ndof
4     for j = 1:Ndof
5         vi = v{i}; uj = u{j};
6         K(:,s) = area.*sum(ww.*cc.*vi.*uj,2);
7         s = s+1;
8     end
9 end

```

现在考虑一个常用情形, 即

$$\int_{\Omega} a \nabla v \cdot \nabla u dx = \int_{\Omega} a(v_x u_x + v_y u_y) dx.$$

这种向量式的积分也可类似前面处理. 设 $\vec{v}_i = [\partial_x \phi_i, \partial_y \phi_i]$, $\vec{u}_j = [\partial_x \phi_j, \partial_y \phi_j]$, 则

$$w_p \cdot (\vec{v}_i \cdot \vec{u}_j) = w_p \partial_x \phi_i \partial_x \phi_j + w_p \partial_y \phi_i \partial_y \phi_j. \quad (8.2)$$

基函数的梯度如下计算

```

1 grad1 = [eta(:,1), -xi(:,1)]./repmat(2*area,1,2);
2 grad2 = [eta(:,2), -xi(:,2)]./repmat(2*area,1,2);
3 grad3 = [eta(:,3), -xi(:,3)]./repmat(2*area,1,2);
4 grad1 = repmat(grad1,1,nG);
5 grad2 = repmat(grad2,1,nG);
6 grad3 = repmat(grad3,1,nG);

```

这里, 梯度也复制了 n_G 列, 从而 gradi 有 $2n_G$ 列. 记 $v_i = \text{gradi}$, $u_j = \text{gradj}$, 令 $k_{ij} = v_i \cdot u_j$, 则给出的矩阵是梯度分量点乘的结果. 由 (8.2), 梯度分量的乘积都要乘以权重, 为此将权重对应 k_{ij} 处理获得权重矩阵.

```

1 ww = zeros(1,2*ng);
2 ww(1:2:end) = weight; ww(2:2:end) = weight;
3 ww = repmat(ww,NT,1);

```

而

$$w_p \cdot (a\vec{v}_i \cdot \vec{u}_j) = w_p a \partial_x \phi_i \partial_x \phi_j + w_p a \partial_y \phi_i \partial_y \phi_j,$$

系数矩阵可类似权重处理.

```

1 cgrad = ones(NT,2*ng);
2 cgrad(:,1:2:end) = cc; cgrad(:,2:2:end) = cc; cc = cgrad;
3 for i = 1:Ndof
4     for j = 1:Ndof
5         vi = v{i}; uj = u{j};
6         K(:,s) = area.*sum(ww.*cgrad.*vi.*uj,2);
7         s = s+1;
8     end
9 end

```

注 8.2 对 v, u 位于不同空间的情形, 只要相应修改基函数及局部自由度个数即可.

基函数, 基函数导数及梯度的程序命名为 Base2D.m, 程序如下

```

1 function w = Base2D(wStr,node,elem,Vh,quadOrder)
2
3 if nargin == 3, Vh = []; quadOrder = 3; end % default: P1
4 if nargin == 4, quadOrder = 3; end
5
6 wStr = lower(wStr); % lowercase string
7 NT = size(elem,1);
8
9 % area
10 z1 = node(elem(:,1),:); z2 = node(elem(:,2),:); z3 = node(elem(:,3),:);
11 xi = [z2(:,1)-z3(:,1), z3(:,1)-z1(:,1), z1(:,1)-z2(:,1)];
12 eta = [z2(:,2)-z3(:,2), z3(:,2)-z1(:,2), z1(:,2)-z2(:,2)];
13 area = 0.5*(xi(:,1).*eta(:,2)-xi(:,2).*eta(:,1));
14
15 % Gauss quadrature rule
16 [lambda,weight] = quadpts(quadOrder); nG = length(weight);
17
18 % gradbasis
19 Dlambdax = eta./repmat(2*area,1,3);
20 Dlambday = -xi./repmat(2*area,1,3);
21 Dlambda1 = [Dlambdax(:,1), Dlambday(:,1)];

```

```

22 Dlambdax = [Dlambdax(:,2), Dlambday(:,2)];
23 Dlambdax = [Dlambdax(:,3), Dlambday(:,3)];
24
25 %% P1-Lagrange
26 if strcmpi(Vh, 'P1') || isempty(Vh)
27 % u.val
28 if contains(wStr, '.val')
29     w1 = repmat(lambda(:,1)', NT, 1); % phi1 at zp, p = 1,2, ...
30     w2 = repmat(lambda(:,2)', NT, 1);
31     w3 = repmat(lambda(:,3)', NT, 1);
32 end
33 % u.dx
34 if contains(wStr, '.dx')
35     w1 = Dlambdax(:,1); w1 = repmat(w1, 1, nG);
36     w2 = Dlambdax(:,2); w2 = repmat(w2, 1, nG);
37     w3 = Dlambdax(:,3); w3 = repmat(w3, 1, nG);
38 end
39 % u.dy
40 if contains(wStr, '.dy')
41     w1 = Dlambday(:,1); w1 = repmat(w1, 1, nG);
42     w2 = Dlambday(:,2); w2 = repmat(w2, 1, nG);
43     w3 = Dlambday(:,3); w3 = repmat(w3, 1, nG);
44 end
45 % u.grad
46 if contains(wStr, '.grad')
47     w1 = Dlambdax1; w1 = repmat(w1, 1, nG);
48     w2 = Dlambdax2; w2 = repmat(w2, 1, nG);
49     w3 = Dlambdax3; w3 = repmat(w3, 1, nG);
50 end
51
52 w = {w1, w2, w3};
53 end

```

这里, `contains(wStr, '.val')` 的意思是, 若 `wStr` 对应的字符串含有 `.val`, 则执行相应语句 (函数 `contains.m` 在 MATLAB 的早期版本中是没有的, 例如 MATLAB R2015a. 为此, 我们设计了 `mycontains.m` 函数用以处理低版本情形). 根据前面的说明, 基函数一般有 `u.val`, `u.dx`, `u.dy`, `u.grad` 等形式, 分别对应函数值和所有一阶导数值.

8.2.2 变分形式的程序设计

现在, 我们逐步给出函数 `assem2d.m` 的内容.

首先是必要的信息, 如

```

1 function varargout = assem2d(Th, Coef, Test, Trial, Vh, quadOrder)
2
3 % Vh --> (v,u)
4
5 %% ----- Preparation for the input -----

```

```

6 % default para
7 if nargin == 4 % default: (v,u) --> (P1,P1)
8     Vh = {'P1','P1'}; quadOrder = 3;
9 end
10 if nargin == 5, quadOrder = 3; end
11
12 % " v = u "
13 if iscell(Vh) && length(Vh)==1 % {'P1'}
14     Vh = repmat(Vh, 1, 2);
15 end
16 if ~iscell(Vh) % 'P1'
17     Vh = repmat( {Vh}, 1, 2);
18 end
19
20 % Gauss-Quadrature
21 [lambda,weight] = quadpts(quadOrder);
22 weight = weight(:)'; % must be a row vector
23 nG = length(weight);
24 % area
25 node = Th.node; elem = Th.elem2D; NT = size(elem,1);
26 z1 = node(elem(:,1,:)); z2 = node(elem(:,2,:)); z3 = node(elem(:,3,:));
27 xi = [z2(:,1)-z3(:,1), z3(:,1)-z1(:,1), z1(:,1)-z2(:,1)];
28 eta = [z2(:,2)-z3(:,2), z3(:,2)-z1(:,2), z1(:,2)-z2(:,2)];
29 area = 0.5*(xi(:,1).*eta(:,2)-xi(:,2).*eta(:,1));

```

当 v, u 位于同一个函数空间时, 我们容许输入为单个符号, 如 $Vh = 'P1'$, 在 assem2d.m 内部扩展为两个.

接着给出稀疏装配指标

```

1 %% ----- Sparse assembling index of Pk-Lagrange element -----
2 % elementwise d.o.f.s
3 [elem2dofv,Ndofv,NNdofv] = dof2d(Th,Vh{1}); % v
4 [elem2dofu,Ndofu,NNdofu] = dof2d(Th,Vh{2}); % u
5
6 % assembling index
7 nnz = NT*Ndofv*Ndofu;
8 ii = zeros(nnz,1); jj = zeros(nnz,1); id = 0;
9 for i = 1:Ndofv
10     for j = 1:Ndofu
11         ii(id+1:id+NT) = elem2dofv(:,i);
12         jj(id+1:id+NT) = elem2dofu(:,j);
13         id = id + NT;
14     end
15 end

```

这里为了程序的简洁, 局部整体对应 elem2dof 由单独的函数 dof2d.m 给定.

双线性形式的装配过程如下

```
1 %% ----- Bilinear form -----
```

```

2 if ~isempty(Trial)
3     K = zeros(NT,Ndofv*Ndofu);
4     for ss = 1:length(Test) % ss-th component of bilinear form
5         % Test and Trial functions (val, dx, dy, grad)
6         v = Test{ss}; u = Trial{ss};
7         vbase = Base2D(v,node,elem,Vh{1},quadOrder); % v1,v2,v3
8         ubase = Base2D(u,node,elem,Vh{2},quadOrder); % u1,u2,u3
9         % Coef matrix
10        cf = Coef{ss};
11        if isnumeric(cf) && length(cf)==1, cf = @(pz) cf+0*pz(:,1); end
12        cc = zeros(NT,nG);
13        for p = 1:nG
14            pz = lambda(p,1)*z1 + lambda(p,2)*z2 + lambda(p,3)*z3;
15            cc(:,p) = cf(pz);
16        end
17        if mycontains(v,'.grad') % of course u = u.grad
18            cgrad = ones(NT,2*nG);
19            cgrad(:,1:2:end) = cc; cgrad(:,2:2:end) = cc; cc = cgrad;
20        end
21        % Weight matrix
22        ww = repmat(weight,NT,1);
23        if mycontains(v,'.grad') % of course u = u.grad
24            ww = zeros(1,2*nG);
25            ww(1:2:end) = weight; ww(2:2:end) = weight;
26            ww = repmat(ww,NT,1);
27        end
28        % Stiffness matrix
29        s = 1;
30        for i = 1:Ndofv
31            for j = 1:Ndofu
32                vi = vbase{i}; uj = ubase{j};
33                K(:,s) = K(:,s) + area.*sum(ww.*cc.*vi.*uj,2);
34                s = s+1;
35            end
36        end
37    end
38    varargout{1} = sparse(ii,jj,K(:,NNdofv,NNdofu)); % kk
39    varargout{2} = K(:,_);
40    return; % The remaining code will be neglected.
41 end

```

注意, 这里对向量型的双线性形式要单独处理. 输出的第一个参数为刚度矩阵 kk , 为了需要, 可以额外输出拉直矩阵

$$[k_{11}, k_{12}, k_{13}, k_{21}, k_{22}, k_{23}, k_{31}, k_{32}, k_{33}].$$

在上面的程序中, 我们添加了 `if` 条件, 以判断是否有试探函数. 如果 `Trial = []`, 那么默认考虑的是如下的线性形式

$$\ell(v) = \int_{\mathcal{T}_h} f v dx.$$

对应用法如下:

```
1 Coef = pde.f; Test = 'v.val';
2 ff = assem2d(Th,Coef,Test,[]);
```

通过加入

```
1 if nargin==3 % Trial = [] --> linear form
2     Trial = [];
3     Vh = {'P1','P1'}; quadOrder = 3;
4 end
```

第 4 个空输入可以去掉. 装配程序中 `return` 的作用是: 当考虑的是双线性形式的时候, 不再执行后面的语句. 而后面的语句就是计算线性形式的, 补充的程序如下

```
1 %% ----- Linear form -----
2 % % isempty(Trial)
3 % Test functions (v.val)
4 v = Test;
5 vbase = Base2D(v,node,elem,Vh{1},quadOrder); % v1,v2,v3
6 % Coef matrix
7 f = Coef;
8 if isnumeric(f) && length(f)==1, f = @(pz) f+0*pz(:,1); end % a constant
9 cc = zeros(NT,nG);
10 for p = 1:nG
11     pz = lambda(p,1)*z1 + lambda(p,2)*z2 + lambda(p,3)*z3;
12     cc(:,p) = f(pz);
13 end
14 % Weight matrix
15 ww = repmat(weight,NT,1);
16 % Load vector
17 F = zeros(NT,Ndofv); % straighten
18 for j = 1:Ndofv
19     vj = vbase{j};
20     F(:,j) = area.*sum(ww.*cc.*vj,2); % [f*phi1, f*phi2, f*phi3]
21 end
22 varargout{1} = accumarray(elem2dofv(:, F(:, [NNdofv 1])), % ff
23 varargout{2} = F(:);
```

8.3 int2d 函数

我们直接编写向量情形的, 它只是在标量情形的基础上添加循环而已.

8.3.1 组合型配对的处理

以线弹性边值问题的第二种变分形式为例进行说明. 双线性形式和右端线性形式分别为

$$a(\mathbf{v}, \mathbf{u}) = 2\mu \int_{\Omega} \varepsilon_{ij}(\mathbf{v}) \varepsilon_{ij}(\mathbf{u}) dx + \lambda \int_{\Omega} (\partial_i v_i)(\partial_j u_j) dx$$

和

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

先考虑双线性形式第一项中的

$$\int_{\Omega} \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx.$$

注意到

$$\varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) = v_{1,x} u_{1,x} + v_{2,y} u_{2,y} + \frac{1}{2} (v_{1,y} + v_{2,x})(u_{1,y} + u_{2,x}), \quad (8.3)$$

我们希望实现如下的对应

```

1 Coef = { 1, 1, 0.5 };
2 Test = {'v1.dx', 'v2.dy', 'v1.dy + v2.dx'};
3 Trial = {'u1.dx', 'u2.dy', 'u1.dy + u2.dx'};
4 A = int2d(Th, Coef, Test, Trial, Vh, quadOrder);

```

显然线性组合的配对 '`v1.dy + v2.dx`' 和 '`u1.dy + u2.dx`' 在计算过程中还需要交叉相乘得

```

1 cvu = { 1, 1, 1, 1 };
2 strv = {'v1.dy', 'v1.dy', 'v2.dx', 'v2.dx'};
3 stru = {'u1.dy', 'u2.dx', 'u1.dy', 'u2.dy'};

```

从而获得新的 `Coef`, `Test` 和 `Trial` (注意组合配对的系数要乘以原来的系数). 从应用的角度来说, 组合配对前的系数一般与 PDE 的数据相关, 但线性组合的系数通常是常数, 本文也只考虑这种情形.

线性组合的配对项数不必相同, 为此考虑如下的配对

```

1 strv = 'v1.dx + 2*v2.val';
2 stru = '3*u1.val + 2*u2.dx + u2.val';

```

1. 实现交叉相乘的第一步是根据 '+' 进行字符拆分. 考虑到空格问题, 我们首先去除 `strv` 和 `stru` 中的空格.

```

1 % remove spaces
2 strv = strrep(strv, ' ', '');
3 stru = strrep(stru, ' ', '');

```

接着进行字符拆分

```
1 % pairs: split on '+'
2 strv = strsplits(strv, '+');
3 stru = strsplits(stru, '+');
```

结果为

```
1 strv = {'v1.dx', '2*v2.val'};
2 stru = {'3*u1.val', '2*u2.dx', 'u2.val'};
```

2. 为了获得交叉相乘的三元组 Coef, Test, Trial, 我们根据 '*' 进行字符拆分.

```
1 % triples: split on '*'
2 nv      = length(strv);      nu      = length(stru);
3 cvv     = cell(1,nv);        cuu     = cell(1,nu);
4 strvv  = cell(1,nv);        struu  = cell(1,nu);
5 for i = 1:nv
6     str = strv{i}; str = strsplits(str, '*');
7     if length(str)==1, cvv{i} = 1; strvv{i} = str{1}; end
8     if length(str)==2, cvv{i} = str2double(str{1}); strvv{i} = str{2}; end
9 end
10 for i = 1:nu
11    str = stru{i}; str = strsplits(str, '*');
12    if length(str)==1, cuu{i} = 1; struu{i} = str{1}; end
13    if length(str)==2, cuu{i} = str2double(str{1}); struu{i} = str{2}; end
14 end
```

这里, cvv 存储 strv 中各组份的系数, strvv 存储去除系数的字符. 当组份中无 '*' 时, 字符串不会拆分, 因而个数为 1, 此时系数为 1; 当有 '*' 时, 字符串拆分为 2 个, 第一个为系数, 第二个为去除系数的字符.

3. 最后交叉组合即可获得三元组.

```
1 % cross pairs: Coef, Test, Trial
2 % Coef
3 cvv = repmat(cvv, 1, nu);
4 Coef1 = cvv([1:2:end, 2:2:end]);
5 Coef2 = repmat(cuu, 1, nv);
6 Coef = cellfun(@(x,y) x*y, Coef1, Coef2, 'UniformOutput', false);
7 % Test
8 strvv = repmat(strvv, 1, nu);
9 Test = strvv([1:2:end, 2:2:end]);
10 % Trial
11 Trial = repmat(struu, 1, nv);
```

以上的说明编写为函数文件

```
1 [Coef,Test,Trial] = getvarForm(strv, stru);
```

有了该函数, (8.3) 对应的完全三元组如下获得.

```

1 Coef = { 1, 1, 0.5 }; % may be function handles
2 Test = {'v1.dx', 'v2.dy', 'v1.dy + v2.dx'};
3 Trial = {'u1.dx', 'u2.dy', 'u1.dy + u2.dx'};
4
5 cc = cell(1,length(Coef)); uu = cc; vv = cc;
6 for ss = 1:length(Test)
7     stru = Trial{ss}; strv = Test{ss};
8     [cs, vs, us] = getvarForm(strv, stru);
9     cf = Coef{ss};
10    for i = 1:length(cs)
11        if isnumeric(cf), cs{i} = cf*cs{i}; end
12        if isa(cf, 'function_handle'), cs{i} = @(p) cf(p)*cs{i}; end
13    end
14    cc{ss} = cs; uu{ss} = us; vv{ss} = vs;
15 end
16
17 Coef = horzcat(cc{:});
18 Test = horzcat(vv{:});
19 Trial = horzcat(uu{:});

```

结果为

```

1 Coef = {1, 1, 0.5, 0.5, 0.5, 0.5};
2 Test = {'v1.dx', 'v2.dy', 'v1.dy', 'v1.dy', 'v2.dx', 'v2.dx'};
3 Trial = {'u1.dx', 'u2.dy', 'u1.dy', 'u2.dx', 'u1.dy', 'u2.dx'};

```

以上讨论总结为函数

```

1 [Coef,Test,Trial] = getExtendedvarForm(Coef,Test,Trial);

```

8.3.2 双线性形式的计算

有了完全三元组, 我们就可以计算双线性形式对应的矩阵了. 这里实际上存在一个问题, 就是会重复计算很多配对. 例如, 第一个和最后一个都是 ('.dx', '.dx') 型的配对, 它们对应相同的矩阵. 注意到系数 Coef 不一定相同, 为了一般性, 暂时不处理该问题.

根据上一小节的说明, 我们已获得了扩展后的三元组 Coef, Test, Trial.

设

$$a(\mathbf{v}, \mathbf{u}) = a\left([v_1, v_2]^T, [u_1, u_2]^T\right) = (\mathbf{v}_1, \mathbf{v}_2)^T \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix},$$

这里, \mathbf{v}_i 的长度为 m_i , \mathbf{u}_i 的长度为 n_i ($m_i = n_i$). Vh 按照 $[v_1, v_2]$ 或 $[u_1, u_2]$ 给出, 如

```

1 Vh = { 'P1', 'P1' };

```

A_{ij} 对应的空间对为

```
1 Vhij = { Vh{i}, Vh{j} }; % FE space pair w.r.t. (vi,uj)
```

向量方程的装配指标为

```
1 %% ----- Sparse assembling index of Pk-Lagrange element -----
2 % elementwise d.o.f.s
3 elem2dofv = cell(1,nSpace); NNdofv = zeros(1,nSpace);
4 for i = 1:nSpace
5     [elem2dofv{i},~,NNdofv(i)] = dof2d(Th,Vh{i}); % vi
6 end
7 elem2dofu = elem2dofv; NNdofu = NNdofv;
8 NNdofvv = sum(NNdofv); NNdofuu = NNdofvv;
9
10 info.NNdofu = NNdofu; %
11
12 % assembling index
13 ii = cell(nSpace^2,1); jj = ii; s = 1;
14 for i = 1:nSpace
15     for j = 1:nSpace
16         [iiv,jju] = getSparse(elem2dofv{i}, elem2dofu{j});
17         ii{s} = iiv + (i>2)*sum(NNdofv(1:i-1));
18         jj{s} = jju + (j>2)*sum(NNdofu(1:j-1));
19         s = s+1;
20     end
21 end
```

这里的 getSparse.m 用以获得块矩阵 A_{ij} 对应的稀疏装配指标 i_{ij}, j_{ij} .

接着, 我们要获得分块刚度矩阵的每一个块对应的 ss. 例如, ss11 如下计算

```
1 % (v1,u1)
2 id11 = contains(Test,'1') & contains(Trial,'1');
3 Coef11 = Coef(id11); Trial11 = Trial(id11); Test11 = Test(id11);
4 [~, ss11] = assem2d(Th,Coef11,Trial11,Test11,Vh,quadOrder);
```

注意, 在 assem2d 中我们新增了第二个输出 K(:,), 其中 K 是装配中给出的行拉直存储矩阵, 如 $K = [k_{11}, k_{12}, k_{13}, k_{21}, k_{22}, k_{23}, k_{31}, k_{32}, k_{33}]$. 上面一段代码中, id11 是获得 (v1, u1) 型的所有配对, 接着按照标量情形计算. 类似可获得其他配对. 下面用循环实现.

```
1 %% ----- Bilinear form -----
2 if isempty(Trial)
3     idvu = true(nSpace^2,1); ss = cell(nSpace^2,1); k = 1;
4     % (vi,uj), i,j = 1,...,nSpace
5     for i = 1:nSpace
6         for j = 1:nSpace
7             Vhij = { Vh{i}, Vh{j} }; % FE space pair w.r.t. (vi,uj)
8             % ----- scalar case -----
9             if nSpace==1
10                 [~, ss{k}] = assem2d(Th,Coef,Test,Trial,Vhij,quadOrder);
```

```

11         break;
12     end
13     % ----- vectorized FEM -----
14     id = mycontains(Test, sprintf('%d', i)) & mycontains(Trial, sprintf('%d', j));
15     if ~sum(id) % empty
16         idvu(k) = false; k = k+1; continue;
17     end
18     [~, ss{k}] = assem2d(Th, Coef(id), Test(id), Trial(id), Vhij, quadOrder);
19     k = k+1;
20 end
21
22 ii = ii(idvu); jj = jj(idvu); ss = ss(idvu);
23 ii = vertcat(ii{:}); jj = vertcat(jj{:}); ss = vertcat(ss{:});
24 output = sparse(ii,jj,ss,NNdofvv,NNdofuu);
25 return; % The remaining code will be neglected.
26 end

```

对向量情形, 配对 (v_i, u_j) 通过下标来搜索. 当没有配对时, $A_{ij} = O$, 它可以直接去掉, 因为我们是用 sparse 装配. 去掉的过程用逻辑数组 `idvu` 实现. 注意, 行和列的个数 `NNdofvv` 和 `NNdofuu` 相同, 因为我们规定 $\mathbf{v} = [v_1, \dots]$ 和 $\mathbf{u} = [u_1, \dots]$ 的分量空间一致, 这里只是为了突出 v 和 u .

8.3.3 线性形式的计算

区域上的线性形式一般为

$$\int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx, \quad \int_{\Omega} f_i v_i dx, \quad i = 1, 2. \quad (8.4)$$

对应的载荷向量只需要按标量情形分块计算即可.

```

1 %% ----- Linear form -----
2 ff = zeros(NNdofvv,1);
3
4 % ----- scalar case -----
5 if nSpace==1
6     output = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
7     return; % otherwise, vectorized FEM
8 end
9
10 % ----- vectorized FEM -----
11 % Test --> v.val = [v1.val, v2.val, v3.val]
12 if strcmpi(Test, 'v.val')
13     trf = eye(nSpace); f = Coef;
14     for i = 1:nSpace
15         Coef = @(pz) f(pz)*trf(:, i); Test = sprintf('v%d.val',i);
16         Fi = assem2d(Th,Coef,Test,[],Vh{i},quadOrder);
17         if i==1, id = 1 : NNdofv(1); end
18         if i≥2, id = NNdofv(i-1) + (1:NNdofv(i)); end

```

```

19         ff(id) = ff(id) + Fi;
20     end
21     output = ff;  return;
22 end
23
24 % Test --> 'v1.val' --> {'v1.val'}
25 if ~iscell(Test), Coef = {Coef}; Test = {Test}; end
26
27 % Test --> {'v1.val', 'v3.val'}
28 for s = 1:length(Test)
29     Coefv = Coef{s}; Testv = Test{s};
30     for i = 1:nSpace
31         str = sprintf('v%d.val',i); % vi.val
32         if strcmpi(Testv, str)
33             Fi = assem2d(Th,Coefv,Testv,[],Vh{i},quadOrder);
34             if i==1, id = 1 : NNdofv(1); end
35             if i≥2, id = NNdofv(i-1) + (1:NNdofv(i)); end
36             ff(id) = ff(id) + Fi;
37         end
38     end
39 end
40 output = ff;

```

这里, 向量情形分为两种情形. 第一种对应 (8.4) 的第一式, 第二种对应 (f_i, v_i) 的某些组合.

8.4 assem1d 和 int1d 函数

assem1d.m 与 assem2d.m 类似, 但仍有一些需要注意的地方.

8.4.1 变分形式的计算

考虑仅含一阶导的双线性形式. 对一维问题, 双线性形式一般是如下典型项的组合

$$\int_{\mathcal{T}_h} av'u'dx, \quad \int_{\mathcal{T}_h} avudx, \quad \int_{\mathcal{T}_h} avu'dx, \quad \int_{\mathcal{T}_h} av'u'dx,$$

这里 \mathcal{T}_h 是由若干线段组成的一维单元, 可以是一维、二维和三维空间中的. 以最后一项为例, 令

$$a(v, u) = \int_{\mathcal{T}_h} av'u'dx,$$

这是因为单元刚度矩阵为 (线性元)

$$[K^e] = \int_K a \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix}' [\phi_1, \phi_2] dx = \int_K a \begin{bmatrix} \phi'_1 \phi_1 & \phi'_1 \phi_2 \\ \phi'_2 \phi_1 & \phi'_2 \phi_2 \end{bmatrix} dx,$$

顺序上一致.

设

$$v_1 = \phi'_1, \quad v_2 = \phi'_2, \quad u_1 = \phi_1, \quad u_2 = \phi_2,$$

则

$$[K^e] = \int_K a \begin{bmatrix} v_1 u_1 & v_1 u_2 \\ v_2 u_1 & v_2 u_2 \end{bmatrix} dx =: \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix},$$

式中,

$$k_{ij} = \int_K a v_i u_j dx, \quad K = [x_l, x_r],$$

且积分可采用 Simpson 公式计算. 为了一般性我们采用 Gauss 积分公式.

为了方便, 以下用 x 表示线段的弧长参数, 那么单元 $K = [x_j, x_{j+1}]$ 的两个节点基为

$$\phi_1(x) = \lambda_1(x) = \frac{x_{j+1} - x}{h_j}, \quad \phi_2(x) = \lambda_2(x) \frac{x - x_j}{h_j},$$

这里的 (λ_1, λ_2) 就是线坐标. 导函数为

$$\lambda'_1(x) = -\frac{1}{h_j}, \quad \lambda'_2(x) = \frac{1}{h_j}.$$

积分转化到线坐标下后, 如下计算

$$k_{ij} = \int_K a v_i u_j dx = |K| \int_{\hat{K}} \tilde{a} \tilde{v}_i \tilde{u}_j dt \approx |K| \sum_{p=1}^{n_g} w_p a(x_p) v_i(x_p) u_j(x_p), \quad (8.5)$$

这里的 Guass 点和权重由 iFEM 包给出, 见那里的函数 quadpts1.m. 用法如下

```
1 [lambda, weight] = quadpts1(order);
```

注意返回的是线坐标, 即参考单元上的结果. `lambda` 有两列, 分别对应 λ_1, λ_2 的 Guass 点, `weight` 是行向量. 区间 $[a, b]$ 的第 p 个 Guass 点如下计算

```
1 pz = lambda(p,1)*za + lambda(p,2)*zb;
```

这里的 `za` 和 `zb` 可以是 d 维空间中线段的两个端点坐标 ($d = 1, 2, 3$).

注意, 因是 d 维情形, 单元长度如下计算

```
1 % length
2 za = node(elem1D(:,1),:); zb = node(elem1D(:,2),:);
3 h = sqrt(sum((zb-za).^2,2));
```

这里的 `node` 是 d 维情形的网格剖分节点坐标集. 所有单元的检验函数值如下

```
1 % Gauss quadrature rule
2 quadOrder = 3;
3 [lambda, weight] = quadpts1(quadOrder); ng = length(weight);
4 Dlambda1 = -1./h; v1 = repmat(Dlambda1, 1, ng);
5 Dlambda2 = 1./h; v2 = repmat(Dlambda2, 1, ng);
```

注意一阶导是常数, 这里复制 n_g 列即得所有积分点处的值. 这样, v_i 是一个矩阵, 行对应单元, 列对应积分点.

试探函数 $u_i = \phi_i$ 如下计算

```
1 u1 = repmat(lambda(:,1)', nel, 1);
2 u2 = repmat(lambda(:,2)', nel, 1);
```

注意, λ 的第 i 列对应 λ_i 的所有积分点, 转置复制 nel 行就是所有单元的. 此时, u_j 是矩阵, 行对应单元, 列对应积分点.

先不考虑积分系数, 按照装配的规定, k_{ij} 如下排列

$$K = [k_{11}, k_{12}, k_{21}, k_{22}],$$

其中每个 k_{ij} 是列向量, 对应所有单元的结果. 记 v_i 与 u_j 点乘后所得矩阵为 k_{ij} , 它的每行对应一个单元, 而每列恰好对应积分公式 (8.5) 右端的一个求和项, 只不过未乘以权重. 对应矩阵 k_{ij} , 我们可获得权重矩阵

```
1 ww = repmat(weight, nel, 1);
```

注意权重 $weight$ 是一行向量, 列对应积分点. 复制 nel 行就是所有单元的结果. 这样, 将 ww 和 k_{ij} 点乘, 然后按行求和就是 (8.5) 右端的求和结果.

```
1 k11 = sum(ww.*v1.*u1, 2);
2 k12 = sum(ww.*v1.*u2, 2);
3 k21 = sum(ww.*v2.*u1, 2);
4 k22 = sum(ww.*v2.*u2, 2);
5 K = [k11, k12, k21, k22];
```

最后, 每行还要乘以单元长度

```
1 ndof = 2;
2 K = repmat(h, 1, ndof^2).*K;
```

当然为了方便, 上面的过程可用一个简单的循环实现.

```
1 K = zeros(nel, ndof^2); s = 1;
2 v = {v1, v2}; u = {u1, u2};
3 for i = 1:ndof
4     for j = 1:ndof
5         vi = v{i}; uj = u{j};
6         K(:, s) = h.*sum(ww.*vi.*uj, 2);
7         s = s+1;
8     end
9 end
```

接着, 我们加上积分前的系数, 注意一般是 d 维的, 如二维问题 Robin 边界条件会产生一维的双线性函数. 根据上面的说明, 我们只要获得类似 v_i 或 u_j 这样的系数矩阵即可, 它

的每行对应一个单元, 每列对应一个积分点. 对第 p 个和项, 所有单元的积分点坐标 px 如下

```
1 pz = lambda(p,1)*za + lambda(p,2)*zb;
```

系数矩阵如下 (例如取 $a = a(x, y) = x + y$)

```
1 cf = @(pz) pz(:,1)+pz(:,2);
2 cc = zeros(nel,ng);
3 for p = 1:ng
4     pz = lambda(p,1)*za + lambda(p,2)*zb;
5     cc(:,p) = cf(pz);
6 end
```

这里为了方便处理维数, 函数的参数采用向量型.

这样, 前面的循环可修改为

```
1 K = zeros(nel,ndof^2); s = 1;
2 v = {v1,v2}; u = {u1,u2};
3 for i = 1:ndof
4     for j = 1:ndof
5         vi = v{i}; uj = u{j};
6         K(:,s) = h.*sum(ww.*cc.*vi.*uj,2);
7         s = s+1;
8     end
9 end
```

注 8.3 一般地, 对 $d \geq 2$ 维, 所有系数函数等均采用向量型变量. 另外, 对 v 和 u 位于不同空间的情形, 可类似 assem2d.m 修改, 这里略.

8.4.2 assem1d 函数

考虑变分形式 (4.3), 我们给出如下对应

```
1 Coef = {pde.a, pde.b, pde.c};
2 Test = {'v.dx', 'v.val', 'v.val'};
3 Trial = {'u.dx', 'u.val'};
```

这里, u 对应试探函数 (即数值解对应的函数), v 对应检验函数; $u.\text{val}$ 表示函数本身, $u.\text{dx}$ 表示一阶导. 注意同一位置的三个数据对应双线性形式的一个组份. 双线性形式的函数格式如下

```
1 kk = assem1d(Th,Coef,Test,Trial,Vh,quadOrder);
```

这里 Th 存储 node 和 elem1D .

基函数及导函数程序如下 (在各自函数内部, 我们不加上 1D)

```

1 function w = Base1D(wStr,node,elem,Vh,quadOrder)
2
3 if nargin == 3, Vh = []; quadOrder = 3; end % default: P1
4 if nargin == 4, quadOrder = 3; end
5
6 wStr = lower(wStr); % lowercase string
7 nel = size(elem,1);
8
9 % length
10 za = node(elem(:,1),:); zb = node(elem(:,2),:);
11 h = sqrt(sum((zb-za).^2,2));
12
13 % Gauss quadrature rule
14 [lambda,weight] = quadpts1(quadOrder); ng = length(weight);
15
16 % derivatives of bases
17 Dlambda1 = -1./h;
18 Dlambda2 = 1./h;
19
20 %% P1-Lagrange
21 if strcmpi(Vh, 'P1') || isempty(Vh)
22     % u.val
23     if contains(wStr, '.val')
24         w1 = repmat(lambda(:,1)',nel,1); % phi1 at xp, p = 1,2,...
25         w2 = repmat(lambda(:,2)',nel,1);
26     end
27     % u.dx
28     if contains(wStr, '.dx')
29         w1 = Dlambda1; w1 = repmat(w1,1,ng);
30         w2 = Dlambda2; w2 = repmat(w2,1,ng);
31     end
32     w = {w1,w2};
33 end

```

这里, 基函数和导函数都是矩阵, 行对应单元, 列对应积分点. 函数文件如下

```

1 function varargout = assem1D(Th,Coef,Test,Trial,Vh,quadOrder)
2
3 % Vh --> (v,u) --> ( Vh{1}, Vh{2} )
4
5 %% ----- Preparation for the input -----
6 % default para
7 if nargin==3 % Trial = [] --> linear form
8     Trial = []; Vh = {'P1','P1'}; quadOrder = 3;
9 end
10 if nargin == 4 % default: (v,u) --> (P1,P1)
11     Vh = {'P1','P1'}; quadOrder = 3;
12 end
13 if nargin == 5, quadOrder = 3; end

```

```

14
15 % " v = u "
16 if iscell(Vh) && length(Vh)==1 % {'P1'}
17     Vh = repmat(Vh, 1, 2);
18 end
19 if ~iscell(Vh) % 'P1'
20     Vh = repmat( {Vh}, 1, 2);
21 end
22
23 % 1D mesh information
24 node = Th.node;
25 elem = Th.elem1D; nel = size(elem1D,1);
26
27 % Guass-Quadrature
28 [lambda,weight] = quadpts1(quadOrder); ng = length(weight);
29 % length
30 za = node(elem(:,1),:); zb = node(elem(:,2),:);
31 h = sqrt(sum((zb-za).^2,2));
32
33 %% ----- Sparse assembling index -----
34 % elementwise d.o.f.s
35 [elem2dofv,ndofv,NNdofv] = dof1d(Th,Vh{1}); % v
36 [elem2dofu,ndofu,NNdofu] = dof1d(Th,Vh{2}); % u
37
38 % assembling index
39 nnz = nel*ndofv*ndofu;
40 ii = zeros(nnz,1); jj = zeros(nnz,1); id = 0;
41 for i = 1:ndofv
42     for j = 1:ndofu
43         ii(id+1:id+nel) = elem2dofv(:,i);
44         jj(id+1:id+nel) = elem2dofu(:,j);
45         id = id + nel;
46     end
47 end
48
49 %% ----- Bilinear form -----
50 if ~isempty(Trial)
51     K = zeros(nel,ndofv*ndofu);
52     for ss = 1:length(Test) % ss-th component of bilinear form
53         % Test and Trial functions (val, dx, ...)
54         v = Test{ss}; u = Trial{ss};
55         vbase = Base1D(v,node,elem1D,Vh{1},quadOrder); % v1,v2
56         ubase = Base1D(u,node,elem1D,Vh{2},quadOrder); % u1,u2
57         % Coef matrix
58         cf = Coef{ss};
59         if isnumeric(cf) && length(cf)==1, cf = @(pz) cf+0*pz(:,1); end % a ...
60             constant, x = pz(:,1)
61         cc = zeros(nel,ng);
62         for p = 1:ng
63             pz = lambda(p,1)*za + lambda(p,2)*zb;

```

```

63         cc(:,p) = cf(pz);
64     end
65     % Weight matrix
66     ww = repmat(weight,nel,1);
67     % Stiffness matrix
68     s = 1;
69     for i = 1:ndofv
70         for j = 1:ndofu
71             vi = vbase{i}; uj = ubase{j};
72             K(:,s) = K(:,s) + h.*sum(ww.*cc.*vi.*uj,2);
73             s = s+1;
74         end
75     end
76 end
77 varargout{1} = sparse(ii,jj,K(:,NNdofv,NNdofu)); % kk
78 varargout{2} = K(:);
79 return; % The remaining code will be neglected.
80 end

```

对二维问题的 P2-Lagrange 有限元, 边界上中点自由度的排序要按 Ω_h 的整体来考虑. 为此, Th 添加域 Th.elem1D. 在上面的程序中, 我们添加了 `if` 条件, 以判断是否有试探函数. 如果 `Trial = []`, 那么默认考虑的是如下的线性形式

$$\ell(v) = \int_{\mathcal{T}_h} f v dx.$$

对应用法如下:

```

1 Coef = pde.f; Test = 'v.val';
2 ff = assem1D(Th,Coef,Test,[],Vh,quadOrder);

```

程序中 `return` 的作用是: 当考虑的是双线性形式的时候, 不再执行后面的语句. 而后面的语句就是计算线性形式的, 补充的程序如下

```

1 %% ----- Linear form -----
2 % % if isempty(Trial)
3 % Test functions (v.val)
4 v = Test;
5 vbase = Base1D(v,node,elem1D,Vh{1},quadOrder); % v1,v2
6 % Coef matrix
7 if isnumeric(Coef)&&size(Coef,2)==ng, cc = Coef; end % matrix of size NT*ng
8 if length(Coef)==1 % function handle or a constant
9     cf = Coef;
10    if isnumeric(cf), cf = @(p) cf + 0*p(:,1); end
11    cc = zeros(nel,ng);
12    for p = 1:ng
13        pz = lambda(p,1)*za + lambda(p,2)*zb;
14        cc(:,p) = cf(pz);
15    end

```

```

16 end
17 % Weight matrix
18 ww = repmat(weight,nel,1);
19 % Load vector
20 F = zeros(nel,ndofv); % straighten
21 for j = 1:ndofv
22     vj = vbase{j};
23     F(:,j) = h.*sum(ww.*cc.*vj,2); % [f*phi1, f*phi2]
24 end
25 varargout{1} = accumarray(elem2dofv(:,1), F(:,1:Ndofv)); % ff
26 varargout{2} = F(:,1);

```

另外, 边界积分可能涉及到边界法向量, 它不能或不太容易用一个统一的匿名函数表达. 为此, 我们容许系数是按规定给出的矩阵, 即行对应单元、列对应积分点的矩阵(若有必要的话, 其他类似的系数矩阵都可如此). 这一点在后面的例子中再说明.

8.4.3 int1d 函数

有了 assem1d.m, 类似二维情形, 我们可获得 int1d.m, 直接给出程序.

```

1 function [output,info] = int1d(Th,Coef,Test,Trial,Vh,quadOrder)
2
3 % e.g.
4 % feSpace --> (v1,v2,v3) or (u1,u2,u3) --> { 'P2','P2','P1' }
5 % vi and ui are in the same FE space
6
7 %% ----- Preparation for the input -----
8 % default para
9 if nargin == 3, Trial = []; Vh = {'P1'}; quadOrder = 3; end
10 if nargin == 4, Vh = {'P1'}; quadOrder = 3; end % default: P1
11 if nargin == 5, quadOrder = 3; end
12
13 if ~iscell(Vh), Vh = {Vh}; end % feSpace = 'P1'
14
15 %% ----- extended [Coef,Test,Trial] -----
16 nSpace = length(Vh);
17 if ~isempty(Trial) && nSpace>1
18     [Coef,Test,Trial] = getExtendedvarForm(Coef,Test,Trial);
19 end
20
21 %% ----- Sparse assembling index of Pk-Lagrange element -----
22 % elementwise d.o.f.s
23 elem2dofv = cell(1,nSpace); Ndofv = zeros(1,nSpace);
24 for i = 1:nSpace
25     [elem2dofv{i},~,Ndofv(i)] = dof1d(Th,Vh{i}); % vi
26 end
27 elem2dofu = elem2dofv; Ndofu = Ndofv;
28 Ndofvv = sum(Ndofv); Ndofuu = Ndofvv;
29

```

```

30 info.NNdofu = NNdofu; %
31
32 % assembling index
33 ii = cell(nSpace^2,1); jj = ii; s = 1;
34 for i = 1:nSpace
35     for j = 1:nSpace
36         [iiv,jju] = getSparse(elem2dofv{i}, elem2dofu{j});
37         ii{s} = iiv + (i≥2)*sum(NNdofv(1:i-1));
38         jj{s} = jju + (j≥2)*sum(NNdofu(1:j-1));
39         s = s+1;
40     end
41 end
42
43 %% ----- Bilinear form -----
44 if isempty(Trial)
45     idvu = true(nSpace^2,1); ss = cell(nSpace^2,1); k = 1;
46     % (vi,uj), i,j = 1,...,nSpace
47     for i = 1:nSpace
48         for j = 1:nSpace
49             Vhij = {Vh{i}, Vh{j}}; % FE space pair w.r.t. (vi,uj)
50             % ----- scalar case -----
51             if nSpace==1
52                 [~, ss{k}] = assem1d(Th,Coef,Test,Trial,Vhij,quadOrder);
53                 break;
54             end
55             % ----- vectorized FEM -----
56             id = mycontains(Test,sprintf('%d',i)) & mycontains(Trial,sprintf('%d',j));
57             if ~sum(id) % empty
58                 idvu(k) = false; k = k+1; continue;
59             end
60             [~, ss{k}] = assem1d(Th,Coef(id),Test(id),Trial(id),Vhij,quadOrder);
61             k = k+1;
62         end
63     end
64     ii = ii(idvu); jj = jj(idvu); ss = ss(idvu);
65     ii = vertcat(ii{:}); jj = vertcat(jj{:}); ss = vertcat(ss{:});
66     output = sparse(ii,jj,ss,NNdofvv,NNdofuu);
67     return; % The remaining code will be neglected.
68 end
69
70 %% ----- Linear form -----
71 ff = zeros(NNdofvv,1);
72
73 % ----- scalar case -----
74 if nSpace==1
75     output = assem1d(Th,Coef,Test,Trial,Vh,quadOrder);
76     return; % otherwise, vectorized FEM
77 end
78
79 % ----- vectorized FEM -----

```

```

80 % Test --> v.val = [v1.val, v2.val, v3.val]
81 if strcmpi(Test, 'v.val')
82     trf = eye(nSpace); f = Coef;
83     for i = 1:nSpace
84         Coef = @(pz) f(pz)*trf(:, i); Test = sprintf('v%d.val',i);
85         Fi = assem1d(Th,Coef,Test,[],Vh{i},quadOrder);
86         if i==1, id = 1 : NNdofv(1); end
87         if i≥2, id = NNdofv(i-1) + (1:NNdofv(i)); end
88         ff(id) = ff(id) + Fi;
89     end
90     output = ff; return;
91 end
92
93 % Test --> 'v1.val' --> {'v1.val'}
94 if ~iscell(Test), Coef = {Coef}; Test = {Test}; end
95
96 % Test --> {'v1.val', 'v3.val'}
97 for s = 1:length(Test)
98     Coefv = Coef{s}; Testv = Test{s};
99     for i = 1:nSpace
100        str = sprintf('v%d.val',i); % vi.val
101        if strcmpi(Testv, str)
102            Fi = assem1d(Th,Coefv,Testv,[],Vh{i},quadOrder);
103            if i==1, id = 1 : NNdofv(1); end
104            if i≥2, id = NNdofv(i-1) + (1:NNdofv(i)); end
105            ff(id) = ff(id) + Fi;
106        end
107    end
108 end
109 output = ff;

```

8.5 Dirichlet 边界条件的处理

设线性方程组分块为

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix},$$

Dirichlet 条件无非对应 U_i ($i = 1, 2, 3$) 中的若干个. 处理 Dirichlet 条件的命令如下

```

1 g_D = {[], g_D2, []};
2 u = Applyboundary2D(Th,kk,ff,g_D,Vh);

```

这里, g_D 元素的个数与 U_i ($i = 1, 2, 3$) 对应, 空元胞表示对应的 U_i 无 Dirichlet 条件.

先考虑局部的 U_i , 即看作普通的标量问题. 我们可获得每个 U_i 的 $bddof$ 和 $bdval$, 分别对应 Dirichlet 自由度的位置和数值, 排成一个向量即为最终的结果. 局部 U_i 的 $bddof$ 和

bdval 由如下函数获得.

```
1 function [bddof,bdval] = getbd2D(Th,g_D,Vh)
2 % g_D: Dirichlet function handle
3
4 if nargin==2, Vh = 'P1'; end % default: P1
5
6 % ----- Dirichlet boundary conditions -----
7 bdStruct = Th.bdStruct;
8 eD = bdStruct.eD; elemD = bdStruct.elemD; bdIndexD = bdStruct.bdIndexD;
9
10 node = Th.node; N = size(node,1);
11 %% P1-Lagrange
12 if strcmpi(Vh, 'P1')
13     NNdof = N;
14     id = eD;
15     isBdNode = false(NNdof,1); isBdNode(id) = true;
16     bddof = (isBdNode);
17     pD = node(eD,:);
18     bdval = g_D(pD);
19 end
20
21 %% P2-Lagrange
22 if strcmpi(Vh, 'P2')
23     NE = Th.NE; NNdof = N + NE;
24     id = [eD; bdIndexD + N];
25     isBdNode = false(NNdof,1); isBdNode(id) = true;
26     bddof = (isBdNode);
27     z1 = node(elemD(:,1),:); z2 = node(elemD(:,2),:); zc = (z1+z2)/2;
28     pD = node(eD,:);
29     uD = g_D(pD); uDc = g_D(zc);
30     bdval = [uD; uDc];
31 end
32
33 %% P3-Lagrange
34 if strcmpi(Vh, 'P3')
35     NE = Th.NE; NT = Th.NT; NNdof = N + 2*NE + NT;
36     id = [eD; bdIndexD+N; bdIndexD+N+NE];
37     isBdNode = false(NNdof,1); isBdNode(id) = true;
38     bddof = (isBdNode);
39     z1 = node(elemD(:,1),:); z2 = node(elemD(:,2),:);
40     za = z1+(z2-z1)/3; zb = z1+2*(z2-z1)/3;
41     pD = node(eD,:);
42     uD = g_D(pD); uDa = g_D(za); uDb = g_D(zb);
43     bdval = [uD; uDa; uDb];
44 end
```

注意, bddof 必须是逻辑数组, 从而可直接拼接向量.

接着逐步实现 Applyboundary2D.m. 标量情形直接调用 getbd2D.m.

```

1 function u = Applyboundary2D(Th,kk,ff,g_D,Vh)
2 %Ex1: g_D = pde.g_D; % scalar
3 %Ex2: g_D = { g_D1, g_D2, g_D3 }
4 %Ex3: g_D = { [], [], g_D3 }
5
6 %% ----- input -----
7 if nargin==4, Vh = {'P1'}; end % default: P1
8 if ~iscell(Vh), Vh = {Vh}; end
9 nSpace = length(Vh);
10
11 %% ----- scalar case -----
12 if nSpace==1
13     [bddof,bdval] = getbd2D(Th,g_D,Vh);
14     u = zeros(size(ff)); u(bddof) = bdval;
15     ff = ff - kk*u;
16     freedof = (~bddof);
17     u(freedof) = kk(freedof, freedof)\ff(freedof); % direct solver
18     return; % otherwise, vectorized FEM
19 end

```

如前所述, 向量情形的 bddof 和 bdval 是分量情形的堆叠.

```

1 %% ----- Vectorized FEM -----
2 % initialization
3 bddof = cell(nSpace,1); bdval = cell(nSpace,1); idval = false(nSpace,1);
4 for i = 1:nSpace
5     [~,NNdofu] = dofnum(Th,Vh{i}); % ui
6     bddof{i} = false(NNdofu,1);
7 end
8 % modify i-th cell of bddof and bdval
9 for i = 1:nSpace
10    if ~isempty(g_D{i})
11        [bddof{i},bdval{i}] = getbd2D(Th,g_D{i},Vh{i});
12        idval(i) = true;
13    end
14 end
15 % delete empty cell of bdval
16 bdval = bdval(idval);
17 bddof = vertcat(bddof{:}); bdval = vertcat(bdval{:});
18
19 u = zeros(size(ff)); u(bddof) = bdval;
20 ff = ff - kk*u;
21 freedof = (~bddof);
22 u(freedof) = kk(freedof, freedof)\ff(freedof); % direct solver

```

在上面的程序中, 我们标记了 bdval 中的空数组, 然后删除. 实际上, 这一步可省略, 因为在向量拼接时, 空数组会自然忽略.

第九章 基于变分形式编程的有限元程序示例

9.1 一维问题 Lagrange 有限元

9.1.1 一维问题的一阶 Lagrange 有限元

考虑两点边值问题

$$-au'' + bu' + cu = f(x), \quad 0 < x < L,$$

取精确解为

$$u(x) = \frac{1}{2e}e^{2x} - \frac{1}{2}(1+e^{-1})e^x + \frac{1}{2}.$$

边界条件可以是 Dirichlet 边界条件或 Neumann 边界条件, 但为了保证解的唯一性, 总假设含有 Dirichlet 边界条件. 变分形式为

$$a(v, u) = \ell(v),$$

其中

$$\begin{aligned} a(v, u) &= \int_0^L (av'u' + bvu' + cvu)dx, \\ \ell(v) &= \int_0^L f(x)v(x)dx + au'v|_0^L. \end{aligned}$$

首先, 我们给定网格和边界条件信息如下

```
1 % ----- Mesh and boundary conditions -----
2 a = 0; b = 1;
3 nel = 10; N = nel+1; % numbers of elements and nodes
4 node = linspace(a,b,nel+1)';
5 elem1D = zeros(nel,2); elem1D(:,1) = 1:N-1; elem1D(:,2) = 2:N;
6 Th.node = node; Th.elem1D = elem1D;
```

为了方便, 类似二维问题, 我们额外定义了 setboundary1D.m 以规定边界条件, 实际上就是确定区间的左右端点的边界类型. 函数用法如下

```
1 bdNeumann = 'abs(x-1)<1e-4';
2 bdStruct = setboundary1D(node, elem1D, bdNeumann);
```

bdStruct 是一个结构体, 包含 Dirichlet 和 Neumann 两个域, 它们分别给定相应的节点序号. PDE 的信息由函数文件 pdedata1D.m 给定, 如下调用.

```
1 pde = pdedata1D;
```

结构体 pde 包含的信息如下

```
1 pde = struct('uexact',@uexact, 'f', @f, 'g_D', @g_D, 'Du', @Du, 'a', @a, 'b', @b, 'c', @c);
```

注意, 系数函数和其他函数都以匿名函数的形式给出.

双线性形式 $a(v, u)$ 对应的刚度矩阵如下实现

```
1 % ----- Stiffness matrix -----
2 Coef = {pde.a, pde.b, pde.c};
3 Test = {'v.dx', 'v.val', 'v.val'};
4 Trial = {'u.dx', 'u.val', 'u.val'};
5 kk = int1d(Th,Coef,Test,Trial,Vh,quadOrder);
```

线性形式的第一项如下

```
1 % ----- Load vector -----
2 Coef = pde.f; Test = 'v.val';
3 ff = int1d(Th,Coef,Test,[],Vh,quadOrder);
```

仅考虑 Neumann 边界条件和 Dirichlet 边界条件. Neumann 边界条件, 即

$$au'|_0^L = ag_Nv|_0^L = (ag_Nv)_L \cdot 1 + (ag_Nv)_0 \cdot (-1),$$

注意左侧端点的外法向量为 -1 , 右侧则为 $+1$. 它可看成退化的双线性形式, 这里系数函数为 a . 给定 Neumann 边界点的序号 Neumann, 若它是向量 elem1D(:,1) 中的元素, 则对应左端点, 从而法向量为 -1 , 反之为 $+1$. 这样, Neumann 边界条件如下处理

```
1 % ----- Neumann boundary conditions -----
2 if ~isempty(Neumann)
3     nvec = 1;
4     if find(Th.elem1D(:,1)==Neumann), nvec = -1; end
5     Dnu = pde.Du(node(Neumann,:))*nvec;
6     ff(Neumann) = ff(Neumann) + Coef(node(Neumann,:))*Dnu;
7 end
```

而 Dirichlet 边界条件如下处理

```
1 % ----- Dirichlet boundary conditions -----
2 isBdNode = false(NN dof,1); isBdNode(Dirichlet) = true;
3 bdDof = (isBdNode); freeDof = (~isBdNode);
4 u = zeros(NN dof,1); u(bdDof) = pde.g_D(node(Dirichlet,:));
5 ff = ff - kk*u;
```

处理边界条件的完整程序见 Applyboundary1D.m (显然高阶元的处理是一致的, 不同的是 NN dof 的数值).

以上说明总结为如下函数文件.

```
1 function uh = FEM1D_variational(Th,pde,Vh,quadOrder)
2
```

```

3 % Quadrature orders for int1d and int2d
4 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
5 if nargin==3, quadOrder = 3; end
6
7 % ----- Stiffness matrix -----
8 Coef = {pde.a, pde.b, pde.c};
9 Test = {'v.dx', 'v.val', 'v.val!'};
10 Trial = {'u.dx', 'u.val', 'u.val!'};
11 kk = int1d(Th,Coef,Test,Trial,Vh,quadOrder);
12
13 % ----- Load vector -----
14 Coef = pde.f; Test = 'v.val';
15 ff = int1d(Th,Coef,Test,[],Vh,quadOrder);
16
17 % ----- Boundary value conditions -----
18 uh = Applyboundary1D(Th,kk,ff,pde);

```

对误差分析, 我们编写了 getL2error1D.m 和 getH1error1D.m 两个函数, 用以计算 L^2 和 H^1 误差. 主程序如下

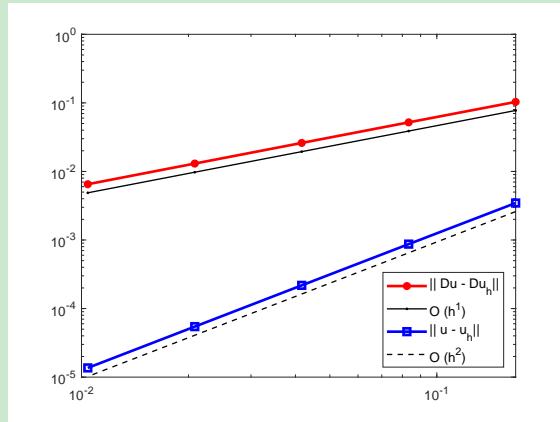
```

1 clc;clear; close all;
2 % ----- Mesh and boundary conditions -----
3 a = 0; b = 1;
4 nel = 3; N = nel+1; % numbers of elements and nodes
5 node = linspace(a,b,nel+1)';
6 elem1D = zeros(nel,2); elem1D(:,1) = 1:N-1; elem1D(:,2) = 2:N;
7
8 bdNeumann = 'abs(x-0)<=1e-4';
9
10 % ----- PDE -----
11 pde = pdedata1D;
12
13 % ----- Poisson -----
14 maxIt = 5;
15 N = zeros(maxIt,1); h = zeros(maxIt,1);
16 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
17
18 Vh = 'P1'; quadOrder = 4;
19 for k = 1:maxIt
20     [node,elem1D] = uniformrefine1D(node,elem1D);
21     bdStruct = setboundary1D(node,elem1D,bdNeumann);
22     Th.node = node; Th.elem1D = elem1D; Th.bdStruct = bdStruct;
23     uh = FEM1D_variational(Th,pde,Vh,quadOrder);
24     nel = size(elem1D,1);
25     h(k) = 1/nel;
26     ErrL2(k) = getL2error1D(node,elem1D,uh,pde.uexact,Vh,quadOrder);
27     ErrH1(k) = getH1error1D(node,elem1D,uh,pde.Du,Vh,quadOrder);
28 end
29
30 % ----- Show rate -----

```

```
31 figure, showrateh(h, ErrL2, ErrH1);
```

误差阶图像如下



可以看到, H^1 和 L^2 的误差阶分别为 1 阶和 2 阶, 这与理论结果相符.

9.1.2 一维问题的高阶 Lagrange 元

现在我们考虑 P2 和 P3 有限元, 程序的修改步骤如下.

Step 1: 在 Base1D.m 中添加高阶元的基函数.

P2-Lagrange 元的局部自由度排列为: 左端点值、右端点值和中点值. 对应的基函数及其导数如下

```
1 %% P2-Lagrange
2 if strcmpi(Vh, 'P2')
3     % u.val
4     if contains(wStr, '.val')
5         w1 = lambda(:,1)'.* (2*lambda(:,1)'-1);    w1 = repmat(w1,nel,1);
6         w2 = lambda(:,2)'.* (2*lambda(:,2)'-1);    w2 = repmat(w2,nel,1);
7         w3 = 4*lambda(:,1)'.*lambda(:,2)';        w3 = repmat(w3,nel,1);
8     end
9     % u.dx
10    if contains(wStr, '.dx')
11        w1 = zeros(nel,ng);   w2 = w1;  w3 = w1;
12        for p = 1:ng
13            w1(:,p) = (4*lambda(p,1)-1)*Dlambda1;
14            w2(:,p) = (4*lambda(p,2)-1)*Dlambda2;
15            w3(:,p) = 4*(lambda(p,1)*Dlambda2+lambda(p,2)*Dlambda1);
16        end
17    end
18    w = {w1,w2,w3};
19 end
```

P3-Lagrange 元的局部自由度排列为: 左端点值、右端点值、 $1/3$ 点值和 $2/3$ 点值. 对应的基函数及其导数如下

```

1 %% P3-Lagrange
2 if strcmpi(Vh, 'P3')
3 % u.val
4 if contains(wStr, '.val')
5     w1 = 0.5*(3*lambda(:,1)-1).*(3*lambda(:,1)-2).*lambda(:,1);
6     w2 = 0.5*(3*lambda(:,2)-1).*(3*lambda(:,2)-2).*lambda(:,2);
7     w3 = 9/2*lambda(:,1).*lambda(:,2).*(3*lambda(:,1)-1);
8     w4 = 9/2*lambda(:,2).*lambda(:,1).*(3*lambda(:,2)-1);
9     w1 = repmat(w1', nel, 1);           w2 = repmat(w2', nel, 1);
10    w3 = repmat(w3', nel, 1);           w4 = repmat(w4', nel, 1);
11 end
12 % u.dx
13 if contains(wStr, '.dx')
14     w1 = zeros(nel, ng);   w2 = w1; w3 = w1;
15     for p = 1:ng
16         w1(:,p) = (27/2*lambda(p,1)^2-9*lambda(p,1)+1).*Dlambda1;
17         w2(:,p) = (27/2*lambda(p,2)^2-9*lambda(p,2)+1).*Dlambda2;
18         w3(:,p) = 9/2*(lambda(p,1)*(3*lambda(p,1)-1).*Dlambda2 ...
19                         + lambda(p,2)*(6*lambda(p,1)-1).*Dlambda1);
20         w4(:,p) = 9/2*(lambda(p,2)*(3*lambda(p,2)-1).*Dlambda1 ...
21                         + lambda(p,1)*(6*lambda(p,2)-1).*Dlambda2);
22     end
23 end
24 w = {w1, w2, w3, w4};
25 end

```

Step 2: 在 assem1d.m 中添加高阶元的局部整体对应矩阵 elem2dof.

elem2dof 由单独的函数文件 dof1d.m 给出, 其用法如下

```

1 % elementwise d.o.f.s
2 [elem2dof, ndof, NNdof] = dof1d(Th, Vh);

```

注意, 该函数本身也用于二维问题边界积分的处理, 而边界积分的计算要匹配二维自由度个数, 所以它与直接的一维问题还是有区别的. 我们通过判断 Th 中是否有 elem 来判断是否是二维问题. P2-Lagrange 对应的程序为

```

1 ndof = 3; NNdof = N + nel;
2 elem2dof = [elem1D, (1:nel)' + N];

```

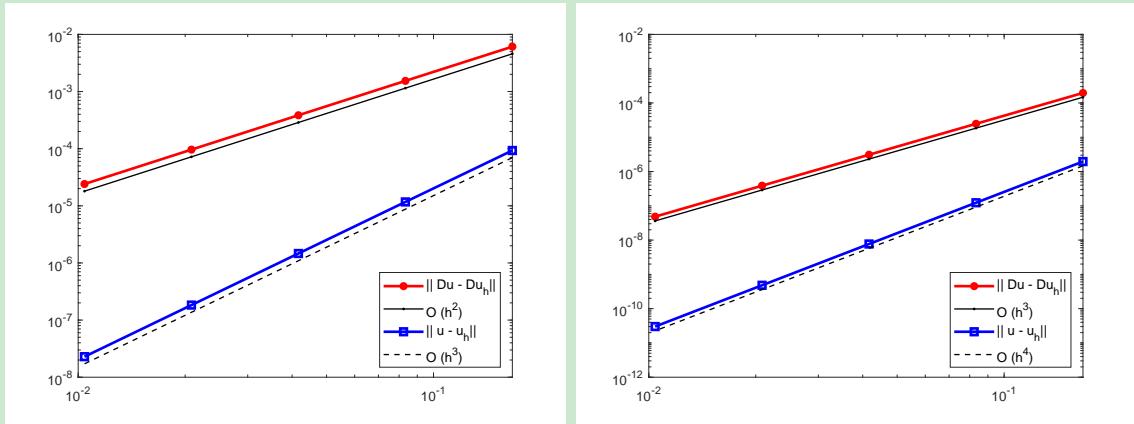
P3-Lagrange 对应的程序为

```

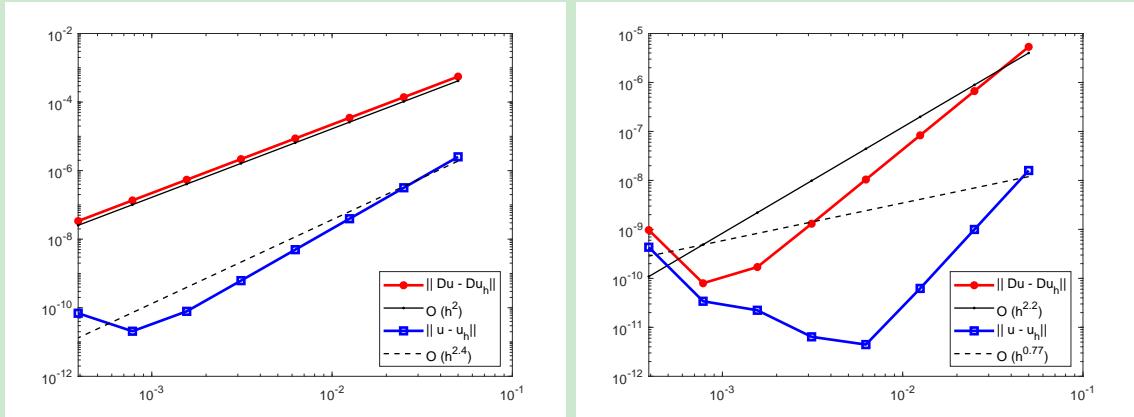
1 ndof = 4; NNdof = N + 2*nel;
2 elem2dof = [elem1D, (1:nel)' + N, (1:nel)' + N + nel];

```

经过上面两步, 高阶元的程序就已经完成了. 为此计算误差阶, 我们还对应修改 getL2error1D.m 和 getH1error1D.m, 这里不再给出其程序. P2-Lagrange 元和 P3-Lagrange 元的误差阶图像如下



可以看到, 误差阶与理论结果相符. 需要指出的是, P1,P2,P3 元的 Gauss 积分阶 quadOrder 分别取为 4,5,6. 前两个元的取为 3,4 即可. 若 P3 元的取为 5, 则所得误差阶都为 4, 即观察到 H^1 误差有超收敛现象. 另外, 初始网格单元个数不宜多, 特别对 3 阶元. 因为此时误差已经很小, 网格加密后误差不一定继续减小, 这是数值方法的共性, 即所谓的半收敛现象, 如下图所示 (初始单元个数为 10)



注: 若从转折处截取线段, 则可发现误差阶与理论结果相符 (注意右侧线段对应的 h 更大).

9.2 二维问题 Lagrange 有限元

9.2.1 二维问题的一阶 Lagrange 有限元

程序编写说明

考虑如下的一般问题

$$\begin{cases} -\nabla \cdot (a \nabla u) + cu = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ g_R u + a \partial_n u = g_N & \text{on } \Gamma_R, \end{cases}$$

设区域 $\Omega = (0, 1)^2$, 且右侧对应 Γ_R , 其他都为 Γ_D . 对齐次 Dirichlet 边界条件, 变分问题为:
找 $u \in V := H_0^1(\Omega)$ 使得

$$a(v, u) = \ell(v), \quad v \in V,$$

式中,

$$\begin{aligned} a(v, u) &= \int_{\Omega} a \nabla v \cdot \nabla u d\sigma + \int_{\Omega} cvud\sigma + \int_{\Gamma_R} g_R vuds, \\ \ell(v) &= \int_{\Omega} fv d\sigma + \int_{\Gamma_R} g_N vds. \end{aligned}$$

初始网格和边界条件如下给出.

```

1 % ----- Mesh and boundary conditions -----
2 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
3 N = 2^1;
4 Nx = N; Ny = N; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node, elem] = squaremesh([a1 b1 a2 b2], h1, h2);
6
7 bdNeumann = 'abs(x-1)<1e-4'; % string for Neumann

```

相关 PDE 数据见 PoissonData2.m, 如下使用.

```

1 % ----- PDE data -----
2 pde = PoissonData2();
3 g_R = @p 1 + p(:,1) + p(:,2); % 1 + x + y
4 pde.g_R = g_R;

```

现在我们来定义函数文件 Poisson_variational.m:

```

1 function u = Poisson_variational(Th, pde, Vh, quadOrder)
2
3 % Quadrature orders for int1d and int2d
4 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
5 if nargin==3, quadOrder = 3; end

```

根据网格 Th 的说明, 我们要添加一维边界积分的网格信息以及辅助数据结构信息.

```

1 % ----- Mesh Th -----
2 % elem1D associated with Gamma_R
3 bdStruct = Th.bdStruct;
4 Th.elem1D = bdStruct.elemN; Th.bdIndex1D = bdStruct.bdIndexN;

```

Ω 上的双线性形式

$$\int_{\Omega} a \nabla v \cdot \nabla u d\sigma + \int_{\Omega} cvud\sigma$$

如下计算.

```

1 % ----- Stiffness matrix -----
2 % Omega

```

```

3 Coef = {pde.a, pde.c};
4 Test = {'v.grad', 'v.val'};
5 Trial = {'u.grad', 'u.val'};
6 kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);

```

Γ_R 上的 elem 要换成相应的一维单元集合, 即 bdStruct.elemN. 而 node 不必更改, 这样也能保证总体刚度矩阵的维数一致. 双线性形式

$$\int_{\Gamma_R} g_R v u ds$$

如下计算.

```

1 % Gamma_R
2 if ~isempty(Th.elem1D)
3     Coef = {pde.g_R};
4     Test = {'v.val'};
5     Trial = {'u.val'};
6     kk = kk + int1d(Th,Coef,Test,Trial,Vh,quadOrder);
7 end

```

接着, 我们考虑线性形式. 右端的

$$\int_{\Omega} f v d\sigma$$

如下计算.

```

1 % ----- Load vector -----
2 % Omega
3 Coef = pde.f; Test = 'v.val';
4 ff = int2d(Th,Coef,Test,[],Vh,quadOrder);

```

边界积分

$$\int_{\Gamma_R} g_N v ds$$

中的区域是 Γ_R , 它由一维的 bdStruct.elemN 给出. 注意到 $g_N = g_R u + a \partial_n u$, 它的表达式必然含边界法向量. 对矩形区域, 边界法向量非常简单. 例如, 考虑右边界, 此时 $n = [1, 0]$, 从而 g_N 可用匿名函数表为

```

1 n = [1,0];
2 g_N = @(p) pde.g_R(p).*pde.uexact(p) + pde.a(p).*(pde.Du(p)*n');

```

我们希望处理一般的多角形区域, 鉴于 $\partial_n u$ 不易用匿名函数表达, int1d.m 的线性部分接受散点值系数, 而这些散点值恰好对应此时的系数矩阵 cc, 它是 nel 行 ng 列的矩阵 (nel 是 elemN 的一维单元个数). assem1d.m 线性部分的系数矩阵如下

```

1 % Coef matrix

```

```

2 if isnumeric(Coef)&&size(Coef,2)==ng, cc = Coef; end % matrix of size NT*ng
3 if length(Coef)==1 % function handle or a constant
4     if isnumeric(Coef), cf = @(p) Coef + 0*p(:,1); end
5     cc = zeros(nel,ng);
6     for p = 1:ng
7         pz = lambda(p,1)*za + lambda(p,2)*zb;
8         cc(:,p) = cf(pz);
9     end
10 end

```

为了方便修改 Robin 或 Dirichlet 边界条件, 我们将定义函数 getMat1d.m, 它计算给定匿名函数 fun 的散点值. 特别地, 程序中单独给出 $\partial_n u$ 对应的矩阵. 该函数如下

```

1 function Cmat = getMat1d(fun,Th1D,bdStruct,quadOrder)
2 if nargin==3, quadOrder = 3; end
3
4 node = Th1D.node; elemN = bdStruct.elemN;
5 nel = size(elemN,1);
6
7 % Gauss-Quadrature
8 [lambda,weight] = quadpts1(quadOrder); ng = length(weight);
9
10 za = node(elemN(:,1),:); zb = node(elemN(:,2),:);
11 Cmat = zeros(nel,ng);
12
13 % function handle is pde.Du
14 z0 = za(1,:); v0 = fun(z0);
15 if length(v0)>1
16     Du = fun;
17     % nvec
18     e = za-zb; he = sqrt(sum(e.^2,2));
19     nvec = [-e(:,2)./he, e(:,1)./he];
20     % Coef matrix: gN
21     for p = 1:ng
22         pz = lambda(p,1)*za + lambda(p,2)*zb;
23         Cmat(:,p) = sum(Du(pz).*nvec,2);
24     end
25     return; % The remaining code will be neglected.
26 end
27
28 % function handle is not pde.Du
29 for p = 1:ng
30     pz = lambda(p,1)*za + lambda(p,2)*zb;
31     Cmat(:,p) = fun(pz);
32 end

```

这里, 通过 `length(v0)` 判断 fun 是否是 `pde.Du` 这样的向量. 这样, 系数矩阵可如下获得,

```

1 % Gamma_R
2 if ~isempty(Th.elem1D)

```

```

3      %Coef = @(p) g_R(p).*pde.uexact(p) + pde.a(p).*(pde.Du(p)*n');
4      Cmat_gR = getMat1d(pde.g_R,Th,quadOrder);
5      Cmat_u = getMat1d(pde.uexact,Th,quadOrder);
6      Cmat_a = getMat1d(pde.a,Th,quadOrder);
7      Cmat_Dnu = getMat1d(pde.Du,Th,quadOrder);
8      Coef = Cmat_gR.*Cmat_u + Cmat_a.*Cmat_Dnu;
9      ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
10 end

```

当然, 这里 $g_R(p) \cdot pde.uexact(p)$ 可直接计算, 而不需要分开计算系数矩阵.

边界条件只剩下 Dirichlet 边界条件, 由函数 Applyboundary2D.m 进行处理.

```

1 % ----- Boundary value conditions -----
2 u = Applyboundary2D(Th,kk,ff,g_D,Vh);

```

程序整理

前面的讨论整理为如下函数

```

1 function u = Poisson_variational(Th,pde,Vh,quadOrder)
2
3 % Quadrature orders for int1d and int2d
4 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
5 if nargin==3, quadOrder = 3; end
6
7 % ----- Mesh Th -----
8 % elem1D associated with Gamma_R
9 bdStruct = Th.bdStruct;
10 Th.elem1D = bdStruct.elemN; Th.bdIndex1D = bdStruct.bdIndexN;
11
12 % ----- Stiffness matrix -----
13 % Omega
14 Coef = {pde.a, pde.c};
15 Test = {'v.grad', 'v.val'};
16 Trial = {'u.grad', 'u.val'};
17 kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
18
19 % Gamma_R
20 if ~isempty(Th.elem1D)
21     Coef = {pde.g_R};
22     Test = {'v.val'};
23     Trial = {'u.val'};
24     kk = kk + int1d(Th,Coef,Test,Trial,Vh,quadOrder);
25 end
26
27 % ----- Load vector -----
28 % Omega
29 Coef = pde.f; Test = 'v.val';
30 ff = int2d(Th,Coef,Test,[],Vh,quadOrder);

```

```

31 % Gamma_R
32 if ~isempty(Th.elem1D)
33     %Coef = @(p) g_R(p).*pde.uexact(p) + pde.a(p).*(pde.Du(p)*n');
34     Cmat_gR = getMat1d(pde.g_R,Th,quadOrder);
35     Cmat_u = getMat1d(pde.uexact,Th,quadOrder);
36     Cmat_a = getMat1d(pde.a,Th,quadOrder);
37     Cmat_Dnu = getMat1d(pde.Du,Th,quadOrder);
38     Coef = Cmat_gR.*Cmat_u + Cmat_a.*Cmat_Dnu;
39     ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
40 end
41
42 % ----- Boundary value conditions -----
43 u = Applyboundary2D(Th,kk,ff,pde,Vh);

```

主程序

主程序可如下编写

```

1 clc;clear;close all;
2 % ----- Mesh and boundary conditions -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 N = 2^1;
5 Nx = N; Ny = N; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
6 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
7
8 bdNeumann = 'abs(x-1)<1e-4'; % string for Neumann
9
10 % ----- PDE data -----
11 pde = Poissondata2();
12 g_R = @(p) 1 + p(:,1) + p(:,2); % 1 + x + y
13 pde.g_R = g_R;
14
15 % ----- Poisson -----
16 maxIt = 5;
17 N = zeros(maxIt,1); h = zeros(maxIt,1);
18 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
19
20 Vh = 'P3';
21 if strcmpi(Vh,'P1'), quadOrder = 3; end
22 if strcmpi(Vh,'P2'), quadOrder = 4; end
23 if strcmpi(Vh,'P3'), quadOrder = 5; end
24 for k = 1:maxIt
25     [node,elem] = uniformrefine(node,elem);
26     %figure(1), showmesh(node,elem), hold on, pause(0.6)
27     Th = getTh(node,elem);
28     uh = Poisson_variational(Th,pde,Vh,quadOrder);
29     NT = size(elem,1);
30     h(k) = 1/sqrt(NT);
31     ErrL2(k) = getL2error(node,elem,uh,pde.uexact,Vh,quadOrder);

```

```

32 ErrH1(k) = getH1error(node, elem, uh, pde.Du, Vh, quadOrder);
33 end
34
35 % ----- Show rate -----
36 figure, showrateh(h, ErrL2, ErrH1);

```

上面的程序计算了 L^2 和 H^1 误差阶.

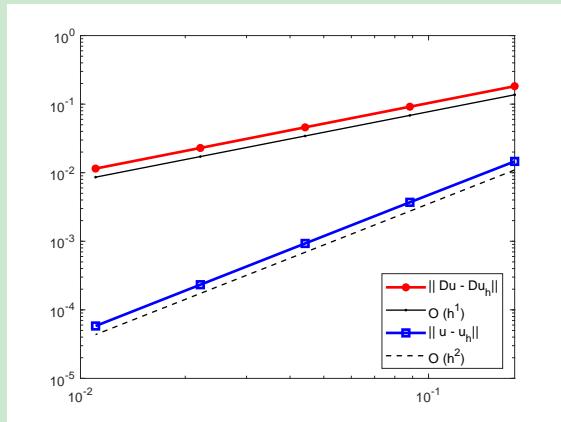
对矩形区域 $\Omega = (0, 1)^2$, 设右边界为 Robin 或 Neumann 边界. 微分方程的信息如下

$$a(x, y) = 1 + x^2 + y^2, \quad c(x, y) = 1, \quad g_R(x, y) = 1 + x + y,$$

精确解取为

$$u(x, y) = \sin(2x + 0.5) \cos(y + 0.3) + \log(3 + xy).$$

误差阶如下图所示.



现在考虑单位圆区域, 可用 MATLAB 的 PDE 工具箱如下生成基本数据结构.

```

1 g = 'circleg';
2 [p,e,t] = initmesh(g, 'hmax', 0.2);
3 node = p'; elem = t(1:3,:)';

```

网格加密可如下进行.

```

1 [p,e,t] = refinemesh(g,p,e,t);

```

取右半圆的边界为 Robin 边界. 微分方程信息不变, 精确解取为 $u = y^2 \sin(\pi x)$, 则主程序可改为

```

1 clc; clear; close all;
2 % ----- Mesh and boundary conditions -----
3 g = 'circleg'; %'lshapeg';
4 [p,e,t] = initmesh(g, 'hmax', 0.5);
5 node = p'; elem = t(1:3,:)';
6
7 bdNeumann = 'x>0'; % string for Neumann

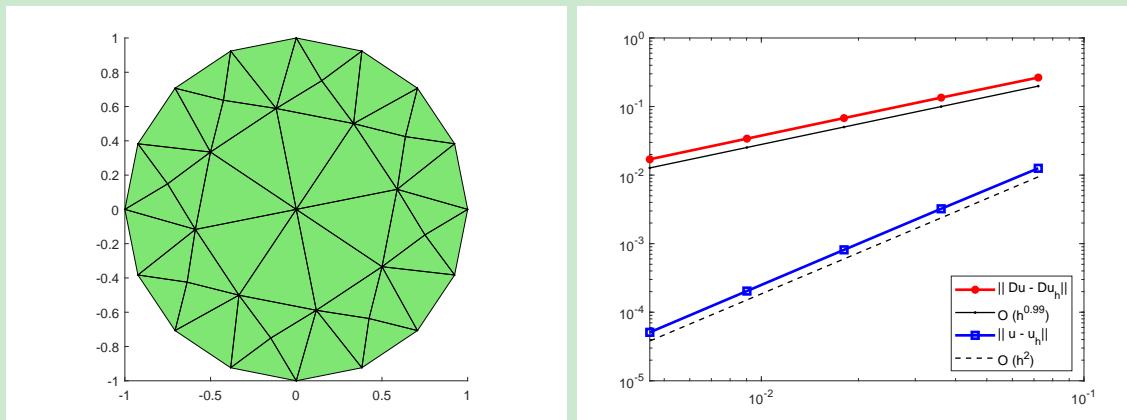
```

```

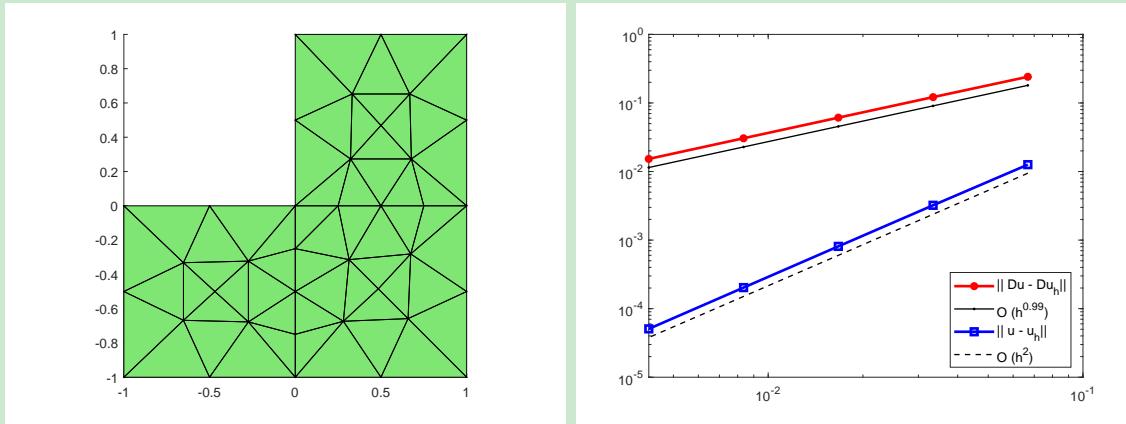
8
9 % ----- PDE data -----
10 pde = Poissonsdata2();
11 g_R = @(p) 1 + p(:,1) + p(:,2); % 1 + x + y
12 pde.g_R = g_R;
13
14 % ----- Poisson -----
15 maxIt = 5;
16 N = zeros(maxIt,1); h = zeros(maxIt,1);
17 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
18
19 Vh = 'P2';
20 if strcmpi(Vh,'P1'), quadOrder = 3; end
21 if strcmpi(Vh,'P2'), quadOrder = 4; end
22 if strcmpi(Vh,'P3'), quadOrder = 5; end
23 for k = 1:maxIt
24     [p,e,t] = refinemesh(g,p,e,t);
25     node = p'; elem = t(1:3,:)';
26     %figure(1), showmesh(node,elem), hold on, pause(0.6)
27     Th = getTh(node,elem);
28     uh = Poisson_variational(Th,pde,Vh,quadOrder);
29     NT = size(elem,1);
30     h(k) = 1/sqrt(NT);
31     ErrL2(k) = getL2error(node,elem,uh,pde.uexact,Vh,quadOrder);
32     ErrH1(k) = getH1error(node,elem,uh,pde.Du,Vh,quadOrder);
33 end
34
35 % ----- Show rate -----
36 figure, showrateh(h,ErrL2,ErrH1);

```

初始网格和误差阶如下图所示。



PDE 工具箱中还预设了 L 型区域 lshapeg。在圆形区域的程序中，把 circleg 替换为 lshapeg，其他条件不变，则初始网格和误差阶的图像如下



9.2.2 二维问题的高阶 Lagrange 元

现在我们考虑 P2 和 P3 有限元, 程序的修改步骤如下.

Step 1: 在 Base1D.m 和 Base2D.m 中添加高阶元的基函数.

Base1D.m 在一维问题中已经处理好, 现在考虑二维的基函数. 二维 P2-Lagrange 元的局部自由度有 6 个, 排列为:

$$v(z_1), v(z_2), v(z_3), v(m_1), v(m_2), v(m_3),$$

其中, z_i 为三角形的顶点, m_i 为 z_i 对应的中点 (注意三角形的第一条边规定为 z_1 的对边). 对应的基函数及其导数如下

```

1 %% P2-Lagrange
2 if strcmpi(Vh, 'P2')
3 % u.val
4 if contains(wStr, '.val')
5 w1 = lambda(:,1)'.*(2*lambda(:,1)'-1);
6 w2 = lambda(:,2)'.*(2*lambda(:,2)'-1);
7 w3 = lambda(:,3)'.*(2*lambda(:,3)'-1);
8 w4 = 4*lambda(:,2)'.*lambda(:,3)';
9 w5 = 4*lambda(:,1)'.*lambda(:,3)';
10 w6 = 4*lambda(:,1)'.*lambda(:,2)';
11 w1 = repmat(w1,NT,1); w2 = repmat(w2,NT,1); w3 = repmat(w3,NT,1);
12 w4 = repmat(w4,NT,1); w5 = repmat(w5,NT,1); w6 = repmat(w6,NT,1);
13 end
14 % u.dx
15 if contains(wStr, '.dx')
16 w1 = zeros(NT,nG); w2 = w1; w3 = w1; w4 = w1; w5 = w1; w6 = w1;
17 for p = 1:nG
18 w1(:,p) = Dlambdax(:,1)*(4*lambda(p,1)-1);
19 w2(:,p) = Dlambdax(:,2)*(4*lambda(p,2)-1);
20 w3(:,p) = Dlambdax(:,3)*(4*lambda(p,3)-1);
21 w4(:,p) = 4*(Dlambdax(:,2)*lambda(p,3) + Dlambdax(:,3)*lambda(p,2));
22 w5(:,p) = 4*(Dlambdax(:,1)*lambda(p,3) + Dlambdax(:,3)*lambda(p,1));
23 w6(:,p) = 4*(Dlambdax(:,1)*lambda(p,2) + Dlambdax(:,2)*lambda(p,1));

```

```

24         end
25     end
26 % u.dy
27 if contains(wStr, '.dy')
28     w1 = zeros(NT,nG); w2 = w1; w3 = w1; w4 = w1; w5 = w1; w6 = w1;
29     for p = 1:nG
30         w1(:,p) = Dlambday(:,1)*(4*lambda(p,1)-1);
31         w2(:,p) = Dlambday(:,2)*(4*lambda(p,2)-1);
32         w3(:,p) = Dlambday(:,3)*(4*lambda(p,3)-1);
33         w4(:,p) = 4*(Dlambday(:,2)*lambda(p,3) + Dlambday(:,3)*lambda(p,2));
34         w5(:,p) = 4*(Dlambday(:,1)*lambda(p,3) + Dlambday(:,3)*lambda(p,1));
35         w6(:,p) = 4*(Dlambday(:,1)*lambda(p,2) + Dlambday(:,2)*lambda(p,1));
36     end
37 end
38 % u.grad
39 if contains(wStr, '.grad')
40     w1 = zeros(NT,2*nG); w2 = w1; w3 = w1; w4 = w1; w5 = w1; w6 = w1;
41     for p = 1:nG
42         w1(:,2*p-1:2*p) = Dlambda1*(4*lambda(p,1)-1);
43         w2(:,2*p-1:2*p) = Dlambda2*(4*lambda(p,2)-1);
44         w3(:,2*p-1:2*p) = Dlambda3*(4*lambda(p,3)-1);
45         w4(:,2*p-1:2*p) = 4*(Dlambda2*lambda(p,3) + Dlambda3*lambda(p,2));
46         w5(:,2*p-1:2*p) = 4*(Dlambda1*lambda(p,3) + Dlambda3*lambda(p,1));
47         w6(:,2*p-1:2*p) = 4*(Dlambda1*lambda(p,2) + Dlambda2*lambda(p,1));
48     end
49 end
50
51 w = {w1,w2,w3,w4,w5,w6};
52 end

```

P3-Lagrange 元的局部自由度有 10 个, 排列为:

$$\begin{cases} v(z_i), & i = 1, 2, 3; \\ v(a_i), & i = 1, 2, 3; \\ v(b_i), & i = 1, 2, 3; \\ v(z_c). \end{cases}$$

这里, a_i 第 i 条边的 $1/3$ 点, b_i 则是 $2/3$ 点, 而 z_c 是单元的重心. 对应的基函数及其导数不再给出.

Step 2: 在 assem1d.m 和 assem2d.m 中添加高阶元的局部整体对应矩阵 elem2dof.

assem1d.m 在一维问题中已处理好, 现考虑 assem2d.m. elem2dof 由单独的函数文件 dof2d.m 给出, 其用法如下

```

1 % elementwise d.o.f.s
2 [elem2dof, Ndof, NNdof] = dof2d(Th, Vh);

```

P2-Lagrange 对应的程序为

```

1 %% P2-Lagrange
2 if strcmpi(Vh, 'P2')
3 % auxstructure
4 auxT = Th.auxT;
5 edge = auxT.edge; NE = size(edge,1);
6 elem2edge = auxT.elem2edge;
7 % d.o.f. numbers
8 Ndof = 6; NNdof = N + NE;
9 % local --> global
10 elem2dof = [elem, elem2edge + N];
11 end

```

P3-Lagrange 对应的程序为

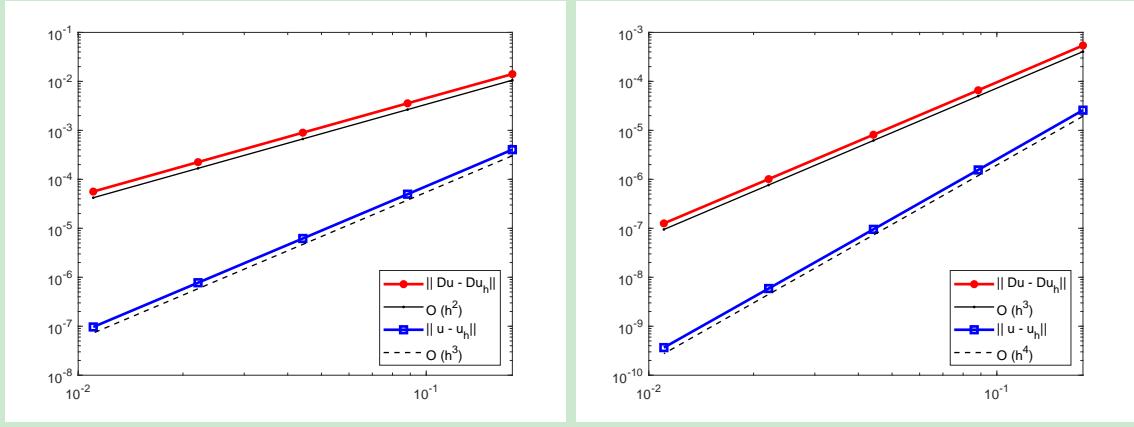
```

1 %% P3-Lagrange
2 if strcmpi(Vh, 'P3')
3 % auxstructure
4 auxT = Th.auxT;
5 edge = auxT.edge; NE = size(edge,1);
6 elem2edge = auxT.elem2edge;
7 % d.o.f. numbers
8 Ndof = 10; NNdof = N + 2*NE + NT;
9 % sgnelem
10 bdStruct = Th.bdStruct;
11 v1 = [2 3 1]; v2 = [3 1 2];
12 bdIndex = bdStruct.bdIndex; E = false(NE,1); E(bdIndex) = 1;
13 sgnelem = sign(elem(:,v2)-elem(:,v1));
14 sgnbd = E(elem2edge); sgnelem(sgnbd) = 1;
15 sgnelem(sgnelem== -1) = 0;
16 elema = elem2edge + N*sgnelem + (N+NE)*(-sgnelem); % 1/3 point
17 elemb = elem2edge + (N+NE)*sgnelem + N*(-sgnelem); % 2/3 point
18 % local --> global
19 elem2dof = [elem, elema, elemb, (1:NT)' + N + 2*NE];
20 end

```

注意, 一维边必须确定好定向再编号, 这里 `sgnelem` 起到该作用. 它是按单元给定的边符号, 若边是正定向 (人为规定的), 则对应 1, 否则对应 0. 为了方便处理边界, 边界边的定向始终规定为逆时针方向.

经过上面两步, 高阶元的程序就已经完成了. 为了计算误差阶, 我们还对应修改 `getL2error.m` 和 `getH1error.m`, 这里不再给出其程序. P2-Lagrange 元和 P3-Lagrange 元的误差阶图像如下



可以看到, 误差阶与理论结果相符. 类似地, 方法也存在半收敛现象, 不再给出.

9.3 线弹性问题的分块编程

分块编程是逐一获得 A_{ij} , 考虑到 (v_i, u_j) 对应的函数空间可不同, 我们约定: 分块编程采用 assem2d.m 和 assem1d.m 实现.

9.3.1 第三种形式的变分问题

程序编写说明

根据线弹性问题的讨论, 双线性形式配对 (v_i, u_j) 对应分块刚度矩阵的 A_{ij} 块, 而每个块都类似 Poisson 方程处理. 针对块结构, 我们可用 int2d.m 等编写线弹性问题的程序. 先考虑第三种形式, 即

$$\begin{aligned} & \mu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx + (\lambda + \mu) \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx \\ & - \mu \int_{\partial\Omega} \partial_n \mathbf{u} \cdot \mathbf{v} ds - (\lambda + \mu) \int_{\partial\Omega} (\operatorname{div} \mathbf{u})(\mathbf{v} \cdot \mathbf{n}) ds = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx. \end{aligned}$$

注意, 该问题仅考虑 Dirichlet 边界条件.

双线性的第一部分可分为

$$\mu \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx \quad \text{和} \quad \mu \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx,$$

它们产生的矩阵相同, 记为 A , 分别对应块 A_{11} 和 A_{22} . 程序如下

```

1 % (v1.grad, u1.grad), (v2.grad, u2.grad)
2 cf = 1;
3 Coef = {cf}; Test = {'v.grad'}; Trial = {'u.grad'};
4 A = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

```

双线性的第二部分可分为

$$\int_{\Omega} v_{1,x} u_{1,x} dx, \quad \int_{\Omega} v_{1,x} u_{2,y} dx, \quad \int_{\Omega} v_{2,y} u_{1,x} dx \quad \text{和} \quad \int_{\Omega} v_{2,y} u_{2,y} dx,$$

它们分别对应块 A_{11} , A_{12} , A_{21} 和 A_{22} . 程序如下

```
1 % (v1.dx, u1.dx)
2 cf = 1;
3 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dx'};
4 B1 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
5 % (v1.dx, u2.dy)
6 cf = 1;
7 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dy'};
8 B2 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
9 % (v2.dy, u1.dx)
10 cf = 1;
11 Coef = {cf}; Test = {'v.dy'}; Trial = {'u.dx'};
12 B3 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
13 % (v2.dy, u2.dy)
14 cf = 1;
15 Coef = {cf}; Test = {'v.dy'}; Trial = {'u.dy'};
16 B4 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

这样, 分块刚度矩阵为

```
1 % kk
2 kk = [ mu*A+(lambda+mu)*B1, (lambda+mu)*B2;
3 (lambda+mu)*B3, mu*A+(lambda+mu)*B4 ];
```

载荷向量的两个部分

$$\int_{\Omega} v_1 f_1 dx \quad \text{和} \quad \int_{\Omega} v_2 f_2 dx$$

分别对应 F_1 和 F_2 , 程序如下

```
1 % F1
2 trf = eye(2);
3 Coef = @(pz) pde.f(pz)*trf(:, 1); Test = 'v.val';
4 F1 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
5
6 % F2
7 trf = eye(2);
8 Coef = @(pz) pde.f(pz)*trf(:, 2); Test = 'v.val';
9 F2 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
10
11 % F
12 ff = [F1; F2];
```

边界条件与 elasticity3.m 的处理相同.

程序整理

以上说明整理为如下函数

```

1 function u = elasticity3_variationalBlock(Th,pde,Vh,quadOrder)
2 %Elasticity3_variationalBlock Conforming P1 elements discretization of linear ...
3 % elasticity equation
4 %
5 %      u = [u1, u2]
6 %      -mu \Delta u - (lambda + mu)*grad(div(u)) = f in \Omega
7 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D.
8 %
9
10
11 % Quadrature orders for int1d and int2d
12 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
13 if nargin==3, quadOrder = 3; end
14
15 % ----- Mesh Th -----
16 node = Th.node; bdStruct = Th.bdStruct; N = size(node,1);
17 mu = pde.mu; lambda = pde.lambda;
18
19 % ----- Stiffness matrix -----
20 % (v1.grad, u1.grad), (v2.grad, u2.grad)
21 cf = 1;
22 Coef = {cf}; Test = {'v.grad'}; Trial = {'u.grad'};
23 A = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
24
25 % (v1.dx, u1.dx) and (v2.dx, u1.dx)
26 cf = 1;
27 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dx'};
28 B1 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
29
30 % (v1.dx, u2.dy) and (v2.dx, u2.dy)
31 cf = 1;
32 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dy'};
33 B2 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
34
35 % kk
36 kk = [ mu*A+(lambda+mu)*B1, (lambda+mu)*B2;
37 (lambda+mu)*B1, mu*A+(lambda+mu)*B2 ];
38
39 % ----- Load vector -----
40 % F1
41 trf = eye(2);
42 Coef = @(pz) pde.f(pz)*trf(:, 1); Test = 'v.val';
43 F1 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
44
45 % F2
46 trf = eye(2);
47 Coef = @(pz) pde.f(pz)*trf(:, 2); Test = 'v.val';
48 F2 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
49

```

```

50 % F
51 ff = [F1; F2];
52
53 % ----- Dirichlet boundary condition -----
54 fixedNode = Th.bdStruct.bdNodeIdx;
55 g_D = pde.g_D; N = Th.N;
56 id = [fixedNode; fixedNode+N];
57 isBdNode = false(2*N,1); isBdNode(id) = true;
58 bdDof = (isBdNode); freeDof = (~isBdNode);
59 pD = Th.node(fixedNode,:);
60 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
61 ff = ff - kk*u;
62
63 % -----
64 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

主程序

主程序如下

```

1 clc;clear;close all
2 % ----- Mesh and boudary conditions -----
3 g = [2 2 2 2 2 2 % decomposed geometry matrix
4     0 1 1 -1 -1 0
5     1 1 -1 -1 0 0
6     0 0 1 1 -1 -1
7     0 1 1 -1 -1 0
8     1 1 1 1 1 1
9     0 0 0 0 0 0];
10 [p,e,t] = initmesh(g,'hmax',1); % initial mesh
11
12 bdNeumann = []; % only Dirichlet condition for elasticity3
13
14 % -----
15 lambda = 1; mu = 1;
16 para.lambda = lambda; para.mu = mu;
17 pde = elasticitydata(para);
18
19 % -----
20 maxIt = 5;
21 N = zeros(maxIt,1); h = zeros(maxIt,1);
22 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
23 errwL2 = zeros(maxIt,1); errwH1 = zeros(maxIt,1);
24 for k = 1:maxIt
25     [p,e,t] = refinemesh(g,p,e,t); % refine mesh
26     node = p'; elem = t(1:3,:)';
27     Th = getTh(node,elem);
28     uh = elasticity3_variational(Th,pde);
29     uh = reshape(uh,[],2);

```

```

30     NT = size(elem,1);      h(k) = 1/sqrt(NT);
31
32     tru = eye(2); trDu = eye(4);
33     errL2 = zeros(1,2);   errH1 = zeros(1,2); % square
34     for id = 1:2
35         uid = uh(:,id);
36         u = @(pz) pde.ueexact(pz)*tru(:, id);
37         Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2:id);
38         errL2(:,id) = getL2error(node, elem, uid, u);
39         errH1(:,id) = getH1error(node, elem, uid, Du);
40     end
41
42     ErrL2(k) = sqrt(sum(errL2.^2,2));
43     ErrH1(k) = sqrt(sum(errH1.^2,2));
44 end
45
46 % ----- Plot convergence rates -----
47 figure;
48 showrateh(h, ErrL2, ErrH1);

```

误差阶图像与 main_elasticity3.m 的结果一致.

9.3.2 第二种形式的变分问题

程序编写说明

双线性形式为

$$a(\mathbf{u}, \mathbf{v}) = 2\mu \int_{\Omega} \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx + \lambda \int_{\Omega} (\partial_i u_i)(\partial_j v_j) dx,$$

右端的线性形式为

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

先考虑双线性形式. 被积函数是

$$v_{i,x}u_{j,x}, \quad v_{i,x}u_{j,y}, \quad v_{i,y}u_{j,x} \quad \text{和} \quad v_{i,y}u_{j,y}, \quad 1 \leq i, j \leq 2$$

的组合, 而且每种求导对应的矩阵是相同的 (即对不同的配对 (i, j)), 它们如下获得.

```

1 % ----- matrices of all pairs -----
2 % (vi.dx, uj.dx)
3 cf = 1;
4 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dx'};
5 A1 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
6 % (vi.dx, uj.dy)
7 cf = 1;
8 Coef = {cf}; Test = {'v.dx'}; Trial = {'u.dy'};
9 A2 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

```

```

10 % (vi.dy, uj.dx)
11 cf = 1;
12 Coef = {cf}; Test = {'v.dy'}; Trial = {'u.dx'};
13 A3 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
14 % (vi.dy, uj.dy)
15 cf = 1;
16 Coef = {cf}; Test = {'v.dy'}; Trial = {'u.dy'};
17 A4 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

```

易知, 第一部分的刚度矩阵如下

```

1 % (Eij(u):Eij(v))
2 A = [ A1 + 0.5*A4,           0.5*A3;
3             0.5*A2           0.5*A1 + A4 ];
4 A = 2*mu*A;

```

第二部分的刚度矩阵如下

```

1 % (div u,div v)
2 B = [ A1,     A2;
3       A3     A4 ];
4 B = lambda*B;

```

最终的刚度矩阵为

```

1 % stiffness matrix
2 kk = A + B;

```

接着计算线性形式. 线性形式的第一项与第三种变分形式一致, 我们考虑第二项

$$\int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

PDE 数据中给定的 g_N 为 $g_N = [\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{12}]$, 令

$$g_1 = [\varepsilon_{11}, \varepsilon_{12}], \quad g_2 = [\varepsilon_{12}, \varepsilon_{22}],$$

则它们相当于 ∇u , 通过截取分量获得. Neumann 边界条件如下处理

```

1 % ----- Neumann boundary condition -----
2 if ~isempty(Th.elem1D)
3     g_N = pde.g_N; trg = eye(3);
4
5     g1 = @(p) g_N(p)*trg(:,[1,3]);
6     Cmat1 = getMat1d(g1,Th,quadOrder);
7     Coef = Cmat1; Test = 'v.val';
8     F1 = F1 + assem1d(Th,Coef,Test,[],Vh,quadOrder);
9
10    g2 = @(p) g_N(p)*trg(:,[3,2]);
11    Cmat2 = getMat1d(g2,Th,quadOrder);

```

```

12     Coef = Cmat2; Test = 'v.val';
13     F2 = F2 + assem1d(Th,Coef,Test,[],Vh,quadOrder);
14 end

```

程序整理

函数文件如下

```

1 function u = elasticity2_variationalBlock(Th,pde,Vh,quadOrder)
2
3 % Quadrature orders for int1d and int2d
4 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
5 if nargin==3, quadOrder = 3; end
6
7 % ----- Mesh Th -----
8 node = Th.node; elem = Th.elem; bdStruct = Th.bdStruct;
9 Th.elem1D = bdStruct.elemN; Th.bdIndex1D = bdStruct.bdIndexN;
10 N = size(node,1);
11 mu = pde.mu; lambda = pde.lambda;
12
13 % ----- matrices of all pairs -----
14 % (vi.dx, uj.dx)
15 cf = 1;
16 Coef = {cf}; Trial = {'u.dx'}; Test = {'v.dx'};
17 A1 = assem2d(Th,Coef,Trial,Test,Vh,quadOrder);
18 % (vi.dx, uj.dy)
19 cf = 1;
20 Coef = {cf}; Trial = {'u.dy'}; Test = {'v.dx'};
21 A2 = assem2d(Th,Coef,Trial,Test,Vh,quadOrder);
22 % (vi.dy, uj.dx)
23 cf = 1;
24 Coef = {cf}; Trial = {'u.dx'}; Test = {'v.dy'};
25 A3 = assem2d(Th,Coef,Trial,Test,Vh,quadOrder);
26 % (vi.dy, uj.dy)
27 cf = 1;
28 Coef = {cf}; Trial = {'u.dy'}; Test = {'v.dy'};
29 A4 = assem2d(Th,Coef,Trial,Test,Vh,quadOrder);
30
31 % ----- Stiffness matrix -----
32 % (Eij(u):Eij(v))
33 A = [ A1 + 0.5*A4,           0.5*A3;
34           0.5*A2           0.5*A1 + A4 ];
35 A = 2*mu*A;
36
37 % (div u,div v)
38 B = [ A1,    A2;
39       A3    A4 ];
40 B = lambda*B;
41

```

```

42 % stiffness matrix
43 kk = A + B;
44
45 % ----- Load vector -----
46 % F1
47 trf = eye(2);
48 Coef = @(pz) pde.f(pz)*trf(:, 1); Test = 'v.val';
49 F1 = assem2d(Th,Coef,[],Test,Vh,quadOrder);
50
51 % F2
52 trf = eye(2);
53 Coef = @(pz) pde.f(pz)*trf(:, 2); Test = 'v.val';
54 F2 = assem2d(Th,Coef,[],Test,Vh,quadOrder);
55
56 % ----- Neumann boundary condition -----
57 if ~isempty(Th.elem1D)
58     g_N = pde.g_N; trg = eye(3);
59
60     g1 = @(p) g_N(p)*trg(:,[1,3]);
61     Cmat1 = getMat1d(g1,Th,quadOrder);
62     Coef = Cmat1; Test = 'v.val';
63     F1 = F1 + assem1d(Th,Coef,[],Test,Vh,quadOrder);
64
65     g2 = @(p) g_N(p)*trg(:,[3,2]);
66     Cmat2 = getMat1d(g2,Th,quadOrder);
67     Coef = Cmat2; Test = 'v.val';
68     F2 = F2 + assem1d(Th,Coef,[],Test,Vh,quadOrder);
69 end
70
71 % load vector
72 ff = [F1; F2];
73
74 % ----- Dirichlet boundary condition -----
75 g_D = pde.g_D; eD = bdStruct.eD;
76 id = [eD; eD+N];
77 isBdNode = false(2*N,1); isBdNode(id) = true;
78 bdDof = (isBdNode); freeDof = (~isBdNode);
79 pD = node(eD,:);
80 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
81 ff = ff - kk*u;
82
83 % ----- Solver -----
84 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

主程序

误差阶与 main_elasticity2.m 的结果一致, 主程序如下

```
1 clc;clear;close all
```

```

2 tic;
3 % ----- Mesh and boundary conditions -----
4 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
5 Nx = 4; Ny = 4; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
6 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
7
8 bdNeumann = 'abs(y-0)<1e-4 | abs(x-1)<1e-4'; % string for Neumann
9
10 % ----- PDE data -----
11 lambda = 1; mu = 1;
12 para.lambda = lambda; para.mu = mu;
13 pde = elasticitydata(para);
14
15 % ----- elasticity1 -----
16 maxIt = 5;
17 N = zeros(maxIt,1); h = zeros(maxIt,1);
18 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
19 errwL2 = zeros(maxIt,1); errwH1 = zeros(maxIt,1);
20 for k = 1:maxIt
21     [node,elem] = uniformrefine(node,elem);
22     Th = getTh(node,elem);
23     uh = elasticity2_variationalBlock(Th,pde);
24     uh = reshape(uh,[],2);
25     NT = size(elem,1); h(k) = 1/sqrt(NT);
26
27     tru = eye(2); trDu = eye(4);
28     errL2 = zeros(1,2); errH1 = zeros(1,2); % square
29     for id = 1:2
30         uid = uh(:,id);
31         u = @(pz) pde.uexact(pz)*tru(:, id);
32         Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2*id);
33         errL2(:,id) = getL2error(node,elem,uid,u);
34         errH1(:,id) = getH1error(node,elem,uid,Du);
35     end
36
37     ErrL2(k) = sqrt(sum(errL2.^2,2));
38     ErrH1(k) = sqrt(sum(errH1.^2,2));
39 end
40
41 % ----- Plot convergence rates -----
42 figure;
43 showrateh(h, ErrL2, ErrH1);
44 toc

```

9.4 双调和方程混合元方法的分块编程

问题说明见节 7.5. 有了线弹性问题的讨论, 这里的过程就比较容易了, 因此不做过多说明.

刚度矩阵如下获得.

```
1 % ----- Stiffness matrix -----
2 % matrix A
3 cf = @(p) 1 + 0*p(:,1);
4 Coef = {cf}; Test = {'v.val'}; Trial = {'u.val'};
5 A = -assem2d(Th,Coef,Test,Trial);
6 % matrix B
7 cf = @(p) 1 + 0*p(:,1);
8 Coef = {cf}; Test = {'v.grad'}; Trial = {'u.grad'};
9 B = assem2d(Th,Coef,Test,Trial);
10 % kk
11 O = zeros(size(B));
12 kk = [A, B, B', O];
```

载荷向量为

```
1 % ----- Load vector -----
2 Coef = pde.f; Test = 'v.val';
3 ff = assem2d(Th,Coef,Test,[]);
4 O = zeros(size(ff));
5 ff = [O; ff];
```

Neumann 边界条件如下处理.

```
1 % ----- Neumann boundary conditions -----
2 Th.elem1D = bdStruct.elemD;
3 %Coef = @(p) pde.Du(p)*n;
4 Coef = getMat1d(pde.Du,Th);
5 ff(1:N) = ff(1:N) + assem1d(Th,Coef,Test,[]);
```

Dirichlet 边界条件如下

```
1 % ----- Dirichlet boundary conditions -----
2 eD = bdStruct.eD; g_D = pde.g_D;
3 id = eD+N;
4 isBdDof = false(2*N,1); isBdDof(id) = true;
5 bdDof = isBdDof; freeDof = (~isBdDof);
6 pD = node(eD,:);
7 U = zeros(2*N,1); U(bdDof) = g_D(pD);
8 ff = ff - kk*U;
9
10 % ----- Solver -----
11 U(freeDof) = kk(freeDof,freeDof)\ff(freeDof);
12 w = U(1:N); u = U(N+1:end);
```

以上讨论汇总后即得函数

```
1 function [u,w] = biharmonicMixedFEM_variationalBlock(Th,pde)
```

主程序略. 数值结果与图 7.3 一致.

9.5 线弹性边值问题的向量编程

经过上面的处理, 我们可以编写向量型问题的 P1,P2 和 P3 元了.

9.5.1 函数文件

考虑第二种形式, 根据前面的说明, 函数文件可如下编写.

```
1 function u = elasticity2_variational(Th,pde,Vh,quadOrder)
2 %Elasticity2_variational
3 % Conforming Lagrange elements of order up to 3 for linear elasticity equation
4 % Variational formulation based programming
5 %      u = [u1, u2]
6 %      -div (sigma) = f in \Omega
7 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D
8 %      Neumann boundary condition \sigma*n = g on \Gamma_N
9 %      \sigma = (\sigma_{ij}): stress tensor, 1≤i,j≤2
10
11 % Quadrature orders for int1d and int2d
12 if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
13 if nargin==3, quadOrder = 3; end
14
15 mu = pde.mu; lambda = pde.lambda;
16
17 % ----- Mesh Th -----
18 % elem1D associated with Gamma_R
19 bdStruct = Th.bdStruct;
20 Th.elem1D = bdStruct.elemN; Th.bdIndex1D = bdStruct.bdIndexN;
21
22 % ----- Stiffness matrix -----
23 % (Eij(u):Eij(v))
24 Coef = { 1, 1, 0.5 };
25 Test = {'v1.dx', 'v2.dy', 'v1.dy + v2.dx'};
26 Trial = {'u1.dx', 'u2.dy', 'u1.dy + u2.dx'};
27 A = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
28 A = 2*mu*A;
29
30 % (div u,div v)
31 Coef = { 1 };
32 Test = { 'v1.dx + v2.dy' };
33 Trial = { 'u1.dx + u2.dy' };
34 B = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
35 B = lambda*B;
36
37 % stiffness matrix
38 kk = A + B;
39
40 % ----- Load vector -----
41 Coef = pde.f; Test = 'v.val';
```

```

42 ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
43
44 % ----- Neumann boundary condition -----
45 if ~isempty(Th.elem1D)
46     g_N = pde.g_N; trg = eye(3);
47
48     g1 = @(p) g_N(p)*trg(:,[1,3]); Cmat1 = getMat1d(g1,Th,quadOrder);
49     g2 = @(p) g_N(p)*trg(:,[3,2]); Cmat2 = getMat1d(g2,Th,quadOrder);
50
51     Coef = {Cmat1, Cmat2}; Test = 'v.val';
52     ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
53 end
54
55 % ----- Dirichlet boundary condition -----
56 tru = eye(2);
57 g_D1 = @(pz) pde.g_D(pz)*tru(:, 1);
58 g_D2 = @(pz) pde.g_D(pz)*tru(:, 2);
59 g_D = {g_D1, g_D2};
60 u = Applyboundary2D(Th,kk,ff,g_D,Vh);

```

9.5.2 主程序

误差阶如下计算.

```

1 clc;clear;close all
2 % ----- Mesh and boudary conditions -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 Nx = 4; Ny = 4; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
6
7 bdNeumann = 'abs(y-0)<1e-4 | abs(x-1)<1e-4'; % string for Neumann
8
9 % ----- PDE data -----
10 lambda = 1; mu = 1;
11 para.lambda = lambda; para.mu = mu;
12 pde = elasticitydata(para);
13
14 % ----- elasticity1 -----
15 maxIt = 5;
16 N = zeros(maxIt,1); h = zeros(maxIt,1);
17 ErrL2 = zeros(maxIt,1); ErrH1 = zeros(maxIt,1);
18 errwL2 = zeros(maxIt,1); errwH1 = zeros(maxIt,1);
19
20 Vh = 'P1';
21 if strcmpi(Vh,'P1'), quadOrder = 3; end
22 if strcmpi(Vh,'P2'), quadOrder = 4; end
23 if strcmpi(Vh,'P3'), quadOrder = 5; end
24 VhNew = repmat( {Vh}, 1, 2 ); % v = [v1,v2]
25 for k = 1:maxIt

```

```

26 [node, elem] = uniformrefine(node, elem);
27 bdStruct = setboundary(node, elem, bdNeumann);
28 Th = getTh(node, elem);
29 uh = elasticity2_variational(Th, pde, VhNew, quadOrder);
30 uh = reshape(uh, [], 2);
31 NT = size(elem, 1); h(k) = 1/sqrt(NT);
32
33 tru = eye(2); trDu = eye(4);
34 errL2 = zeros(1, 2); errH1 = zeros(1, 2); % square
35 for id = 1:2
36 uid = uh(:, id);
37 u = @(pz) pde.uexact(pz)*tru(:, id);
38 Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2:id);
39 errL2(:, id) = getL2error(node, elem, uid, u, Vh, quadOrder);
40 errH1(:, id) = getH1error(node, elem, uid, Du, Vh, quadOrder);
41 end
42
43 ErrL2(k) = sqrt(sum(errL2.^2, 2));
44 ErrH1(k) = sqrt(sum(errH1.^2, 2));
45 end
46
47 % ----- Plot convergence rates -----
48 figure;
49 showrateh(h, ErrL2, ErrH1);

```

误差阶图像如下

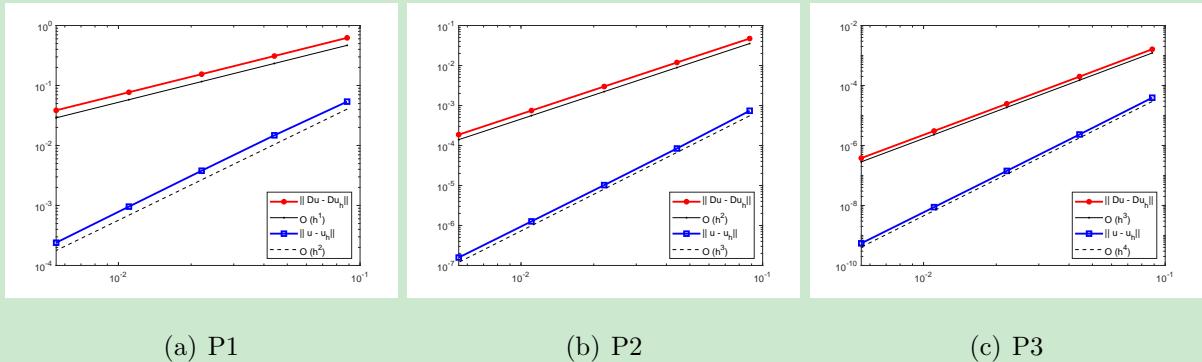


图 9.1. 线弹性问题的误差阶 (P1,P2,P3-Lagrange 元)

9.6 双调和方程混合元方法的向量编程

混合元方法也可看成向量型问题, 本节用基于变分形式的程序进行计算. 考虑问题

$$\begin{cases} \Delta^2 u = f & \text{in } \Omega \subset \mathbb{R}^2, \\ u = g_1, \partial_n u = g_2 & \text{on } \partial\Omega. \end{cases}$$

令 $w = -\Delta u$, 混合变分问题为: 找 $(w, u) \in H^1(\Omega) \times H_0^1(\Omega)$ 使得

$$\begin{cases} -\int_{\Omega} w\phi dx + \int_{\Omega} \nabla u \cdot \nabla \phi dx = \int_{\partial\Omega} \partial_n u \phi ds, & \phi \in H^1(\Omega), \\ \int_{\Omega} \nabla w \cdot \nabla \psi dx = \int_{\Omega} f \psi dx, & \psi \in H_0^1(\Omega). \end{cases}$$

记 $u_1 = w, u_2 = u$ 以及 $v_1 = \phi, v_2 = \psi$, 则上面的问题可视为以 $\mathbf{u} = (u_1, u_2)^T$ 为解, $\mathbf{v} = (v_1, v_2)^T$ 为检验函数的向量型变分问题, 即

$$\begin{cases} -\int_{\Omega} u_1 v_1 dx + \int_{\Omega} \nabla u_2 \cdot \nabla v_1 dx = \int_{\partial\Omega} g_2 v_1 ds, & v_1 \in H^1(\Omega), \\ \int_{\Omega} \nabla u_1 \cdot \nabla v_2 dx = \int_{\Omega} f v_2 dx, & v_2 \in H_0^1(\Omega). \end{cases}$$

易知, 它等价于求和后的变分问题

$$\int_{\Omega} (-v_1 u_1 + \nabla v_1 \cdot \nabla u_2 + \nabla v_2 \cdot \nabla u_1) dx = \int_{\Omega} f v_2 dx + \int_{\partial\Omega} g_2 v_1 ds.$$

双线性形式对应的矩阵如下计算.

```

1 % ----- Stiffness matrix -----
2 Coef = { -1, 1, 1 };
3 Trial = {'u1.val', 'u2.grad', 'u1.grad'};
4 Test = {'v1.val', 'v1.grad', 'v2.grad'};
5 kk = int2d(Th,Coef,Trial,Test,Vh,quadOrder);

```

f 所在线性形式的载荷向量为

```

1 % ----- Load vector -----
2 Coef = pde.f; Test = 'v2.val';
3 ff = int2d(Th,Coef,[],Test,Vh,quadOrder);

```

Neumann 条件对应的载荷向量为

```

1 % ----- Neumann boundary conditions -----
2 % 1D-mesh
3 bdStruct = Th.bdStruct;
4 Th.elem1D = bdStruct.elemD; Th.bdIndex1D = bdStruct.bdIndexD;
5 % Coef = @(p) pde.Du(p)*n;
6 Coef = getMat1d(pde.Du,Th,quadOrder); Test = 'v1.val';
7 ff = ff + int1d(Th,Coef,[],Test,Vh,quadOrder);

```

Dirichlet 边界条件如下处理.

```

1 % ----- Dirichlet boundary conditions -----
2 g_D = { [], pde.g_D };
3 U = Applyboundary2D(Th,kk,ff,g_D,Vh);

```

数值解为

```

1 U = reshape(U, [], 2);
2 w = U(:, 1); u = U(:, 2);

```

函数文件与主程序略, 误差阶结果如下.

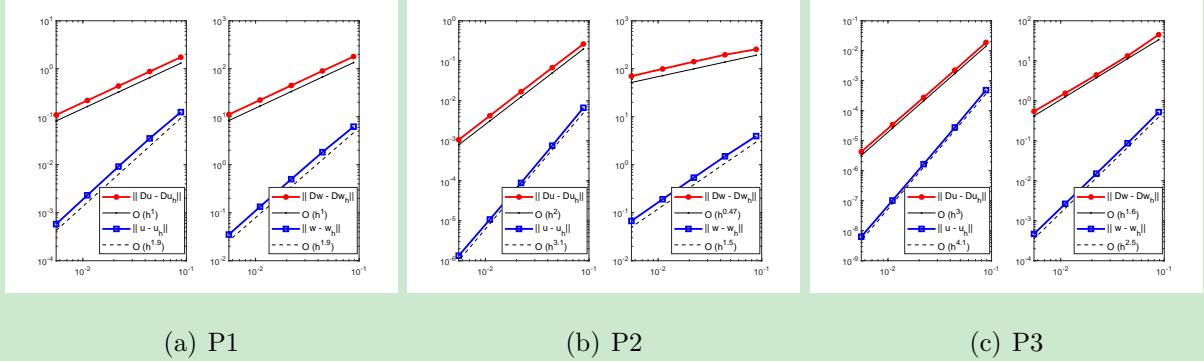


图 9.2. 双调和方程混合元方法的误差阶 (P1,P2,P3-Lagrange 元)

可以看到, u 的误差阶都是最优的, 但 w 对后两者是次优的 (与 Δu 的阶差不多).

9.7 Stokes 方程

9.7.1 变分问题

Stokes 问题为

$$\begin{cases} -\nu \Delta \mathbf{u} + \nabla p = \mathbf{f} & \text{in } \Omega, \\ -\nabla \cdot \mathbf{u} = 0 & \text{in } \Omega, \\ \mathbf{u} = \mathbf{g} & \text{on } \partial\Omega, \end{cases} \quad (9.1)$$

其中, $\Omega \subset \mathbb{R}^d$ ($d = 2, 3$), $\mathbf{u} = (u_1, \dots, u_d)$ 为流体速度, p 为流体压力, $\nu = 1/R_e > 0$ (R_e 是 Reynolds 数).

- Models the steady state flow of viscous fluid
- Conservation of mass: $\nabla \cdot \mathbf{u} = 0$ (“incompressibility condition”)

在 (9.1) 的第一式两边点乘向量 $\mathbf{v} \in H_0^1(\Omega)^d$, 并积分得

$$\int_{\Omega} -\nu \Delta \mathbf{u} \cdot \mathbf{v} dx + \int_{\Omega} \nabla p \cdot \mathbf{v} dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx.$$

分部积分, 有

$$\int_{\Omega} \sum_{i=1}^d \nu \nabla u_i \cdot \nabla v_i dx - \int_{\Omega} p \sum_{i=1}^d \frac{\partial v_i}{\partial x_i} dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx, \quad (9.2)$$

其中, $\mathbf{v} = (v_1, \dots, v_d)$.

若记

$$\nabla \mathbf{u} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \dots & \frac{\partial u_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_d}{\partial x_1} & \dots & \frac{\partial u_d}{\partial x_d} \end{bmatrix}, \quad \nabla \mathbf{v} = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \dots & \frac{\partial v_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial v_d}{\partial x_1} & \dots & \frac{\partial v_d}{\partial x_d} \end{bmatrix},$$

且对 d 阶矩阵 $A = (a_{ij})$ 和 $B = (b_{ij})$, 定义矩阵间的内积

$$A : B = \sum_{i=1}^d \sum_{j=1}^d a_{ij} b_{ij},$$

则

$$\sum_{i=1}^d \nabla u_i \cdot \nabla v_i = \nabla \mathbf{u} : \nabla \mathbf{v}.$$

于是 (9.2) 可写为

$$\int_{\Omega} \nu \nabla \mathbf{u} : \nabla \mathbf{v} dx - \int_{\Omega} (\operatorname{div} \mathbf{v}) p dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx,$$

式中,

$$\operatorname{div} \mathbf{v} = \sum_{i=1}^d \frac{\partial v_i}{\partial x_i} = \nabla \cdot \mathbf{v}.$$

综上, 变分问题为: 找 $\mathbf{u} \in H^1(\Omega)^d$ 且 $\mathbf{u}|_{\partial\Omega} = \mathbf{g}$, 以及 $p \in L^2(\Omega)/\mathbb{R}$, 使得

$$\begin{cases} \int_{\Omega} \nu \nabla \mathbf{u} : \nabla \mathbf{v} dx - \int_{\Omega} (\operatorname{div} \mathbf{v}) p dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx, & \mathbf{v} \in H_0^1(\Omega)^d, \\ - \int_{\Omega} (\operatorname{div} \mathbf{u}) q dx = 0, & q \in L^2(\Omega)/\mathbb{R}. \end{cases}$$

这里, p 容许相差一个常数, 故在相差一个常数等价的空间 $L^2(\Omega)/\mathbb{R}$ 中唯一, 另一种取法是

$$q \in L_0^2(\Omega) := \left\{ q \in L^2(\Omega) : \int_{\Omega} q dx = 0 \right\}.$$

为了方便, 记

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \nu \nabla \mathbf{u} : \nabla \mathbf{v} dx, \quad b(\mathbf{v}, q) = - \int_{\Omega} (\operatorname{div} \mathbf{v}) p dx, \\ F(\mathbf{v}) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx, \end{aligned}$$

则变分问题可写为: 找 $\mathbf{u} \in H^1(\Omega)^d$ 且 $\mathbf{u}|_{\partial\Omega} = \mathbf{g}$, 以及 $p \in L^2(\Omega)/\mathbb{R}$, 使得

$$\begin{cases} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = F(\mathbf{v}), & \mathbf{v} \in \mathbf{V} = H_0^1(\Omega)^d, \\ b(\mathbf{u}, q) = 0, & q \in P = L_0^2(\Omega). \end{cases}$$

设 \mathcal{T}_h 是 Ω 的正则剖分, 我们考虑协调元空间 $\mathbf{V}_h \subset \mathbf{V}$, $P_h \subset P$. 典型的稳定有限元空间对 (\mathbf{V}_h, P_h) 包括: MINI element, Girault - Raviart element 和 $P_k - P_{k-1}$ 元. 一个特殊的例子是 $P_2 - P_1$ 元, 也称为 Taylor-Hood 元.

有限元问题为: 找 $(\mathbf{u}_h, p_h) \in \mathbf{V}_h \times P_h$, 使得

$$\begin{cases} a(\mathbf{u}_h, \mathbf{v}) + b(\mathbf{v}, p_h) = F(\mathbf{v}), & \mathbf{v} \in \mathbf{V}_h, \\ b(\mathbf{u}_h, q) = 0, & q \in P_h. \end{cases} \quad (9.3)$$

问题 (9.3) 的求解方法很多, 可以直接离散化为方程组 (鞍点问题), 另一种方法是把两者合在一起, 即找 $(\mathbf{u}_h, p_h) \in \mathbf{V}_h \times L^2(\Omega)$, 使得

$$a(\mathbf{u}_h, \mathbf{v}) + b(\mathbf{v}, p_h) + b(\mathbf{u}_h, q) - \varepsilon(p_h, q) = F(\mathbf{v}), \quad \mathbf{v} \in \mathbf{V}_h, \quad q \in L^2(\Omega), \quad (9.4)$$

其中, ε 是充分小的数以保证稳定性 (实际上等价于在第二个方程中加入摄动项). 注意此时 p_h 是唯一解.

9.7.2 分块表示

为了表示更清楚, 我们把问题 (9.1) 的第一式按分量具体写出来, 有 (二维问题)

$$\begin{cases} -\nu \Delta u_1 + \frac{\partial p}{\partial x} = f_1, \\ -\nu \Delta u_2 + \frac{\partial p}{\partial y} = f_2, \end{cases}$$

变分形式的获得实际上就是在第一行和第二行分别乘以 $v_1, v_2 \in H_0^1(\Omega)$, 分部积分后再加起来.

分部积分且未相加之前的结果为

$$\begin{cases} \nu \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx - \int_{\Omega} p \frac{\partial v_1}{\partial x} dx = \int_{\Omega} f_1 v_1 dx, \\ \nu \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx - \int_{\Omega} p \frac{\partial v_2}{\partial y} dx = \int_{\Omega} f_2 v_2 dx. \end{cases}$$

问题 (9.1) 的第二式相应为

$$0 = - \int_{\Omega} (\operatorname{div} \mathbf{u}) q dx = - \int_{\Omega} \frac{\partial u_1}{\partial x} q dx - \int_{\Omega} \frac{\partial u_2}{\partial y} q dx.$$

以上几式简写在一起为

$$\begin{cases} a_1(v_1, u_1) + b_1(v_1, p) = F_1(v_1), \\ a_2(v_2, u_2) + b_2(v_2, p) = F_2(v_2), \\ b_1(q, u_1) + b_2(q, u_2) = 0, \end{cases}$$

添加摄动项后为

$$\begin{aligned} & a_1(v_1, u_1) + a_2(v_2, u_2) \\ & + b_1(v_1, p) + b_2(v_2, p) \\ & + b_1(q, u_1) + b_2(q, u_2) \\ & - \varepsilon(q, p) \\ & = F_1(v_1) + F_2(v_2). \end{aligned}$$

当我们把元素在基函数下表达时, 有如下的矩阵表达

$$\begin{aligned}
& \mathbf{v}_1^T A_1 \mathbf{u}_1 + \mathbf{v}_2^T A_2 \mathbf{u}_2 \\
& + \mathbf{v}_1^T B_1 \mathbf{p} + \mathbf{v}_2^T B_2 \mathbf{p} \\
& + \mathbf{q}^T C_1 \mathbf{u}_1 + \mathbf{q}^T C_2 \mathbf{u}_2 \\
& + \mathbf{q}^T D_\varepsilon \mathbf{p} \\
& = \mathbf{v}_1^T \mathbf{F}_1 + \mathbf{v}_2^T \mathbf{F}_2,
\end{aligned}$$

其中 $C_1 = B_1^T, C_2 = B_2^T$. 上式可写为

$$[\mathbf{v}_1^T, \mathbf{v}_2^T, \mathbf{q}^T] \begin{bmatrix} A_1 & O & B_1 \\ O & A_2 & B_2 \\ C_1 & C_2 & D_\varepsilon \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{p} \end{bmatrix} = [\mathbf{v}_1^T, \mathbf{v}_2^T, \mathbf{q}^T] \begin{bmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \mathbf{0} \end{bmatrix},$$

即

$$\begin{bmatrix} A_1 & O & B_1 \\ O & A_2 & B_2 \\ C_1 & C_2 & D_\varepsilon \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \mathbf{0} \end{bmatrix}.$$

9.7.3 分块编程

函数文件为

```

1 function [uh,ph] = Stokes_variationalBlock(Th,pde,Vh,quadOrder)
2 %Stokes_variationalBlock Taylor-Hood (P2-P1) FEM
3 % variational formulation based programming in terms of block matrix
4 %      u = [u1, u2]
5 %      -div(mu*grad u) + grad p = f in \Omega,
6 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma
7
8 % Quadrature orders for int1d and int2d
9 if nargin==2, Vh = {'P2','P2','P1'}; quadOrder = 3; end % default: Taylor-Hood
10 if nargin==3, quadOrder = 4; end
11
12 % ----- Stiffness matrix -----
13 % A1,A2
14 Coef = { 1 }; Test = {'v1.grad'}; Trial = {'u1.grad'};
15 A1 = assem2d(Th,Coef,Test,Trial,Vh(1),quadOrder);
16 A2 = A1;
17
18 % B1,B2
19 Coef = { -1 }; Test = {'v1.dx'}; Trial = {'p.val'};
20 B1 = assem2d(Th,Coef,Test,Trial,Vh([1,3]),quadOrder);
21 Coef = { -1 }; Test = {'v2.dy'}; Trial = {'p.val'};
22 B2 = assem2d(Th,Coef,Test,Trial, Vh([1,3]), quadOrder);

```

```

23 % note: we cannot use int2d.m to compute B2 since "v = u" (v = [v1,v2,...], 
24 % u = [u1,u2,...]). However, we are allowed to use assem2d.m and 
25 % assem1d.m instead when dealing with the programming in terms of block matrix. 
26 
27 % C1,C2 
28 C1 = B1'; C2 = B2'; 
29 
30 % D 
31 eps = 1e-10; 
32 Coef = { -eps }; Test = {'p.val'}; Trial = {'q.val'}; 
33 D = assem2d(Th,Coef,Test,Trial,Vh(3),quadOrder); 
34 
35 % stiffness matrix 
36 NNdof = size(A1,1); 
37 O = zeros(NNdof,NNdof); 
38 kk = [ A1, O, B1; 
39 O, A2, B2; 
40 C1, C2, D ]; 
41 
42 % ----- Load vector ----- 
43 ff = zeros(size(kk,1),1); 
44 % F1 
45 trf = eye(2); 
46 Coef = @(pz) pde.f(pz)*trf(:, 1); Test = 'v1.val'; 
47 F1 = assem2d(Th,Coef,Test,[],Vh(1),quadOrder); 
48 
49 % F2 
50 trf = eye(2); 
51 Coef = @(pz) pde.f(pz)*trf(:, 2); Test = 'v2.val'; 
52 F2 = assem2d(Th,Coef,Test,[],Vh(2),quadOrder); 
53 
54 ff(1:2*NNdof) = [F1;F2]; 
55 
56 % ----- Dirichlet boundary condition ----- 
57 tru = eye(2); 
58 g_D1 = @(pz) pde.g_D(pz)*tru(:, 1); 
59 g_D2 = @(pz) pde.g_D(pz)*tru(:, 2); 
60 g_D = {g_D1, g_D2, []}; 
61 U = Applyboundary2D(Th,kk,ff,g_D,Vh); 
62 uh = [U(1:NNdof), U(NNdof+1:2*NNdof)]; % uh = [u1h, u2h] 
63 ph = U(2*NNdof+1:end); 

```

主程序如下

```

1 clc;clear;close all 
2 % ----- Mesh and boudary conditions ----- 
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1; 
4 Nx = 2; Ny = 2; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny; 
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2); 
6

```

```

7 % ----- PDE data -----
8 pde = Stokesdata();
9
10 % ----- elasticity1 -----
11 maxIt = 4;
12 N = zeros(maxIt,1); h = zeros(maxIt,1);
13 ErruL2 = zeros(maxIt,1); ErruH1 = zeros(maxIt,1);
14 ErrpL2 = zeros(maxIt,1); ErrpH1 = zeros(maxIt,1);
15
16 Vh = {'P2','P2','P1'}; quadOrder = 4;
17 for k = 1:maxIt
18     [node,elem] = uniformrefine(node,elem);
19     Th = getTh(node,elem);
20     [uh,ph] = Stokes_variationalBlock(Th,pde,Vh,quadOrder);
21     NT = size(elem,1); h(k) = 1/sqrt(NT);
22
23     tru = eye(2); trDu = eye(4);
24     erruL2 = zeros(1,2); erruH1 = zeros(1,2); % square
25     for id = 1:2
26         uid = uh(:,id);
27         u = @(pz) pde.uexact(pz)*tru(:, id);
28         Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2:id);
29         erruL2(:,id) = getL2error(node,elem,uid,u,Vh{1},4);
30         erruH1(:,id) = getH1error(node,elem,uid,Du,Vh{1},4);
31     end
32
33     ErruL2(k) = sqrt(sum(erruL2.^2,2));
34     ErruH1(k) = sqrt(sum(erruH1.^2,2));
35
36     ErrpL2(k) = getL2error(node,elem,ph,pde.pexact,Vh{3},4);
37     ErrpH1(k) = getH1error(node,elem,ph,pde.Dpexact,Vh{3},4);
38 end
39
40 % ----- Plot convergence rates -----
41 figure;
42 showrateh(h, ErruL2, ErruH1, '||u - u_h||', '||Du - Du_h||');
43 figure;
44 showrateh(h, ErrpL2, ErrpH1, '||p - p_h||', '||Dp - Dp_h||');

```

误差阶计算结果如下, 它们都是最优的.

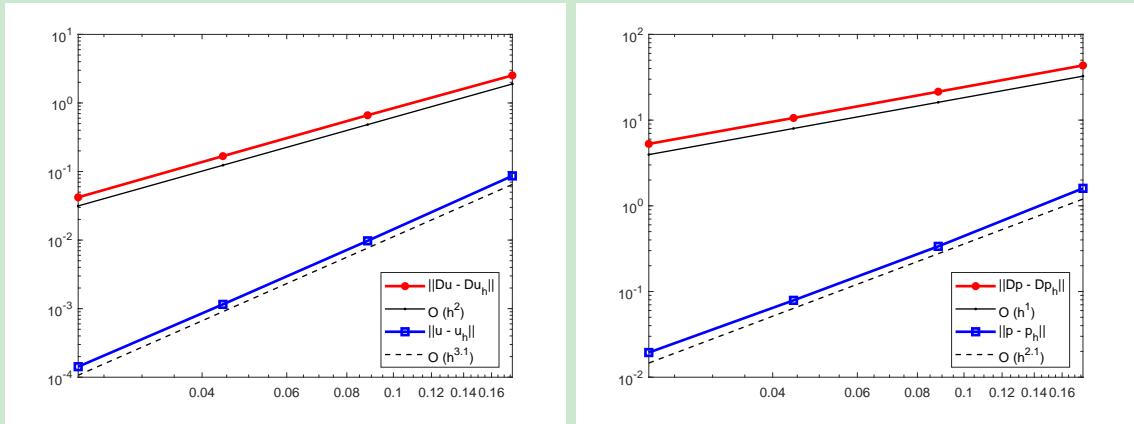


图 9.3. Stokes 方程的误差阶 (Taylor-Hood 元)

9.7.4 向量编程

函数文件为

```

1 function [uh,ph] = Stokes_variational(Th,pde,Vh,quadOrder)
2 %Stokes_variational Taylor-Hood (P2-P1) FEM
3 % variational formulation based programming
4 %      u = [u1, u2]
5 %      -div(mu*grad u) + grad p = f in \Omega,
6 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma
7
8 % Quadrature orders for int1d and int2d
9 if nargin==2
10    Vh = {'P2','P2','P1'}; quadOrder = 4;
11 end % default: Taylor-Hood
12 if nargin==3, quadOrder = 4; end
13
14 % ----- Stiffness matrix -----
15 vstr = {'v1','v2','q'}; ustr = {'u1','u2','p'};
16 % [v1,v2,q] = [v1,v2,v3], [u1,u2,p] = [u1,u2,u3]
17 % a1(v1,u1) + a2(v2,u2)
18 % + b1(v1,p) + b2(v2,p)
19 % + b1(q,u1) + b2(q,u2) - eps*(q,p)
20 eps = 1e-10;
21 Coef = { 1, 1, -1,-1, -1,-1, -eps};
22 Test = {'v1.grad', 'v2.grad', 'v1.dx', 'v2.dy', 'q.val', 'q.val', 'q.val'};
23 Trial = {'u1.grad', 'u2.grad', 'p.val', 'p.val', 'u1.dx', 'u2.dy', 'p.val'};
24 [Test,Trial] = getStdvarForm(vstr, ustr, Test, Trial);
25 [kk,info] = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
26
27 % ----- Load vector -----
28 trf = eye(2);
29 Coef1 = @(pz) pde.f(pz)*trf(:, 1); Coef2 = @(pz) pde.f(pz)*trf(:, 2);
30 Coef = {Coef1, Coef2};
31 Test = {'v1.val', 'v2.val'};

```

```

32 ff = int2d(Th,Coeff,Test,[],Vh,quadOrder);
33
34 % ----- Dirichlet boundary condition -----
35 tru = eye(2);
36 g_D1 = @(pz) pde.g_D(pz)*tru(:, 1);
37 g_D2 = @(pz) pde.g_D(pz)*tru(:, 2);
38 g_D = {g_D1, g_D2, []};
39 U = Applyboundary2D(Th,kk,ff,g_D,Vh);
40 NNdofu = info.NNdofu;
41 id1 = NNdofu(1); id2 = NNdofu(1)+NNdofu(2);
42 uh = [ U(1:id1), U(id1+1:id2) ]; % uh = [u1h, u2h]
43 ph = U(id2+1:end);

```

注意, 我们额外编写了函数 getStdvarForm.m, 它把 $[u_1, u_2, p]$ 转化为 $[u_1, u_2, u_3]$.

主程序如下

```

1 clc;clear;close all
2 % ----- Mesh and boudary conditions -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 Nx = 2; Ny = 2; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
6
7 % ----- PDE data -----
8 pde = Stokesdata();
9
10 % ----- elasticity1 -----
11 maxIt = 5;
12 N = zeros(maxIt,1); h = zeros(maxIt,1);
13 ErruL2 = zeros(maxIt,1); ErruH1 = zeros(maxIt,1);
14 ErrpL2 = zeros(maxIt,1); ErrpH1 = zeros(maxIt,1);
15
16 Vh = {'P2','P2','P1'}; % v = [v1,v2,q] --> [v1,v2,v3]
17 quadOrder = 4;
18 for k = 1:maxIt
19     [node,elem] = uniformrefine(node,elem);
20     Th = getTh(node,elem);
21     [uh,ph] = Stokes_variational(Th,pde,Vh,quadOrder);
22     NT = size(elem,1); h(k) = 1/sqrt(NT);
23
24     tru = eye(2); trDu = eye(4);
25     erruL2 = zeros(1,2); erruH1 = zeros(1,2); % square
26     for id = 1:2
27         uid = uh(:,id);
28         u = @(pz) pde.ueexact(pz)*tru(:, id);
29         Du = @(pz) pde.Du(pz)*trDu(:, 2*id-1:2*id);
30         erruL2(:,id) = getL2error(node,elem,uid,u,Vh{1},4);
31         erruH1(:,id) = getH1error(node,elem,uid,Du,Vh{1},4);
32     end
33
34     ErruL2(k) = sqrt(sum(erruL2.^2,2));

```

```
35 ErruH1(k) = sqrt(sum(erruH1.^2,2));
36
37 ErrpL2(k) = getL2error(node,elem,ph,pde.pexact,Vh{3},4);
38 ErrpH1(k) = getH1error(node,elem,ph,pde.Dpexact,Vh{3},4);
39 end
40
41 % ----- Plot convergence rates -----
42 figure;
43 showrateh(h, ErruL2, ErruH1, '|||u - u_h|||', '|||Du - Du_h|||');
44 figure;
45 showrateh(h, ErrpL2, ErrpH1, '|||p - p_h|||', '|||Dp - Dp_h|||');
```

误差阶的数值结果与分块编程相同.

第十章 基于变分形式的三维有限元的程序设计

待添加

三维问题与二维问题类似, 考虑到前面未处理过三维问题. 我们单独讨论三维问题, 并对前面基于变分形式的处理过程进行一个回顾.

10.1 assem3d

10.1.1 节点基与 Gauss 求积公式

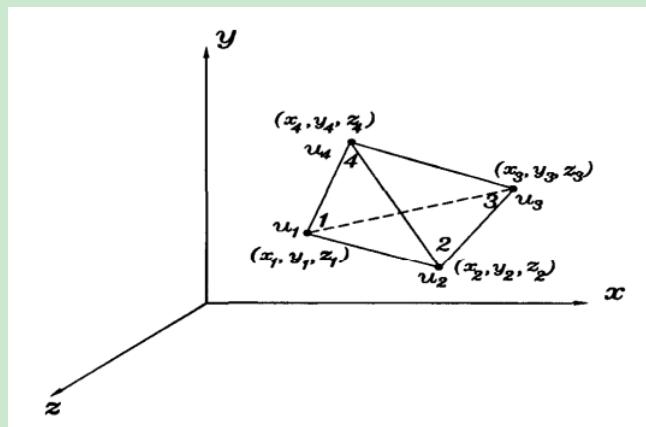


图 10.1. 四面体单元

对三维问题, 我们使用四面体作为单元. 如图 10.1 所示, 设局部节点基分别为 $\phi_1, \phi_2, \phi_3, \phi_4$, 即

$$u(x, y, z) = u_1\phi_1 + u_2\phi_2 + u_3\phi_3 + u_4\phi_4, \quad (x, y, z) \in K.$$

节点基可用重心坐标获得. 设顶点 P_i 对应的重心坐标函数为 $\phi_i = \lambda_i$, 任取四面体内一点 P , 则

$$\lambda_i(x, y, z) = \frac{\Delta PP_2P_3P_4}{\Delta P_1P_2P_3P_4}.$$

为了方便, 这里仍用 Δ 表示有向体积 (单元的体积可直接用 Δ 表示). 类似地, 重心坐标与直角坐标有如下关系

$$\begin{bmatrix} 1 \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix}.$$

将系数矩阵按第一列的 $x_1, y_1, z_1, 1$ 展开的代入余子式分别记为 a_1, b_1, c_1, d_1 , 类似可定义 a_i, b_i, c_i, d_i , 则有

$$\lambda_i(x, y, z) = (a_i x + b_i y + c_i z + d_i) / \Delta, \quad i = 1, 2, 3, 4.$$

而且

$$\begin{cases} x = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 + \lambda_4 x_4, \\ y = \lambda_1 y_1 + \lambda_2 y_2 + \lambda_3 y_3 + \lambda_4 y_4, \\ z = \lambda_1 z_1 + \lambda_2 z_2 + \lambda_3 z_3 + \lambda_4 z_4, \\ 1 = \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4. \end{cases} \quad (10.1)$$

体积积分的 Gauss 公式与面积情形一样

$$\int_K f(x, y, z) dV = |K| \sum_{p=1}^{n_G} w_p f(x_p, y_p, z_p),$$

这里的积分节点 x_p, y_p, z_p 由参考元上的通过 (10.1) 获得. 参考元上的积分节点和权重由 iFEM 中如下函数给定

```
1 [lambda, weight] = quadpts3(quadOrder);
```

这里 `lambda` 有 4 列 `nG` 行, 每列对应一个基函数的全部积分节点值. 而 `weight` 是一行向量.

第十一章 基于变分形式的有限元迭代格式的程序设计

待添加

11.1 发展方程

11.1.1 抛物型方程

11.1.2 双曲型方程

11.2 非线性问题

Part IV

有限元求解器

第十二章 自适应有限元方法

12.1 跳量积分计算的程序设计

对自适应有限元方法, 一般需要计算如下的跳量积分

$$\sum_{e \subset \partial K} c_e \| [u_h] \|_{0,e}^2, \quad \sum_{e \subset \partial K} c_e \| [\partial_{n_e} u_h] \|_{0,e}^2, \quad K \in \mathcal{T}_h,$$

式中的 c_e 在 e 上为常数, u_h 是有限元方法的数值解, 跳量定义为

$$[u] = u|_{K_1} - u|_{K_2}.$$

跳量积分的计算相当麻烦, 甚至比有限元的函数文件更难设计. 为了方便, 我们将用给定的函数, 如 $u(x, y) = \sin x \sin y$ 进行 P1 协调元插值以确定 u_h . 此时的第一个结果为零.

12.1.1 积分节点的连通性

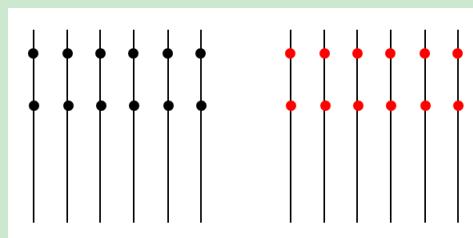
我们始终使用一维的 Gauss 积分计算 e 上的积分, 它不包括端点. 先考虑 u_h , 为了处理一般性的问题, 我们假设 u_h 只能通过局部展开式

$$u_h|_K = u_1 \phi_1 + u_2 \phi_2 + u_3 \phi_3$$

获得, 如 P1 协调元插值. 不失一般性, 这里仅考虑三个局部节点基, 且分别对应三角形的顶点.

1. 边上积分节点的编号

- 积分节点是跳跃点, 对左右两个单元的限制, 我们将采用不同的编号, 就像 DG 方法一样. 为了方便序号的查询与对应, 边界边也给出两种编号, 可以认为其中一种对应零延拓的限制.



- 如图, 黑色竖线表示 `edge` 按自然顺序的排列, 红色是黑色的复制, 对应另一种编号. 在对边上节点进行编号时, 我们始终规定按照 `edge` 的定向进行.

- 将黑色第 1 条边的积分节点编号为 $1 : n_g$, 那么红色第 1 条边的积分节点编号则为 $(1 : n_g) + N_E \cdot n_g$. 也就是说, 红黑边对应节点的编号相差 $N_E \cdot n_g$.

2. 积分节点的连通性

- 首先可生成类似 `elem` 或一般的 `elem2dof` 的连通性矩阵. 它的每行对应一个单元, 每列对应一个自由度. 注意到内部边存在方向问题, 我们规定与 `edge` 同定向的边属于黑色边, 反向的属于红色边, 这只需要事先算出边的符号.

```

1 % auxiliary data
2 totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
3 [edge,i1,totalJ] = unique(totalEdge,'rows');
4 NT = size(elem,1); NE = size(edge,1);
5 elem2edge = reshape(totalJ,NT,3);
6 % sign of elementwise edges
7 sgnElem = sign([elem(:,3)-elem(:,2), elem(:,1)-elem(:,3), ...
    elem(:,2)-elem(:,1)]);

```

有时候我们要恢复边界边的定向, 这是局部插值表达式可能涉及到边界的方向问题. 这里暂时不添加.

- 根据“黑红规定”, 所有单元第 1 条边的编号如下

```

1 ng = 2;
2 % first side
3 ei = elem2edge(:,1); sgn1 = sgnElem(:,1);
4 id = repmat(1:ng, NT,1); id(sgn1<0,:) = (ng:-1:1)+NE*ng;
5 elem2dofe1 = id + repmat((ei-1)*ng,1,ng);

```

对负定向边, 它的节点编号要逆序进行, 以匹配 `edge` 的节点. 类似可给出其他两条边的结果, 合在一起得到连通性矩阵

```

1 % elem2dofe
2 elem2dofe = [elem2dofe1,elem2dofe2,elem2dofe3];

```

注意, 边界边因只出现一次, 故编号也只出现一次, 它可能对应黑色, 也可能对应红色.

12.1.2 插值的计算

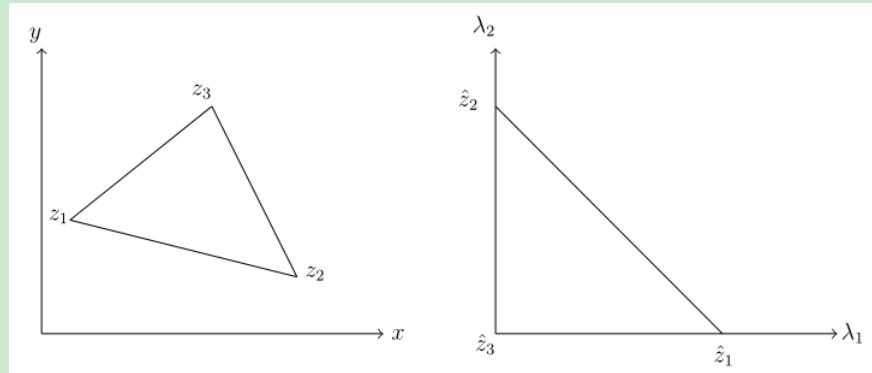


图 12.1. (x, y) 平面到 (λ_1, λ_2) 平面单元三角形的变化

基函数一般由面积坐标确定, 积分节点仿射变换到参考单元上即可得到相应的 $(\lambda_1(p), \lambda_2(p), \lambda_3(p))$ 的值, 这里的 p 表示单元上第 p 个积分节点. 设 $[0, 1]$ 上的积分节点和权重分别为 $r = [r_1, r_2, \dots, r_{n_g}]$ 和 $w = [w_1, w_2, \dots, w_{n_g}]$, 可由 quadpts1 获得

```

1 [lambda1d, weight1d] = quadpts1(4); ng = length(weight1d);
2 [~, id] = sort(lambda1d(:,1));
3 lambda1d = lambda1d(id,:); weight1d = weight1d(id);

```

注意, quadpts1 给出的 λ 不一定是增序, 这里进行了重排. 利用比例关系知, 参考单元上三条边的 (λ_1, λ_2) 坐标分别为

- 第 1 条边: $(0, r_{n_g}), \dots, (0, r_2), (0, r_1)$;
- 第 2 条边: $(r_1, 0), (r_2, 0), \dots, (r_{n_g}, 0)$;
- 第 3 条边: 由关系式 $\lambda_3 = 1 - \lambda_1 - \lambda_2$ 确定.

类似 quadpts.m 的输出, 可如下给出积分节点的 $(\lambda_1, \lambda_2, \lambda_3)$:

```

1 % quadrature points on the reference triangle
2 lambdae1 = [zeros(ng,1), lambda1d(:,2), lambda1d(:,1)];
3 lambdae2 = [lambda1d(:,1), zeros(ng,1), lambda1d(:,2)];
4 lambdae3 = 1 - lambdae1 - lambdae2;
5 lambdaRef = [lambdae1; lambdae2; lambdae3];

```

对 P1 协调元, 每个单元上对应的插值为

```

1 % interpolant at quadrature points
2 elemuh = zeros(NT, 3*ng); Ndof = 3;
3 for p = 1:3*ng
4     % basis functions at the p-th quadrature point

```

```

5      base = lambdaRef(p,:); % phi1,phi2,phi3
6      % interpolation at the p-th quadrature point
7      for i = 1:Ndof
8          elemuh(:,p) = elemuh(:,p) + uh(elem2dof(:,i))*base(:,i);
9      end
10 end

```

对其他类型的有限元, 只要修改 `base` 即可. 对导数插值, 如 $\partial_x u_h$, 要把 `base` 修改为基函数的偏导数. 参考单元上的积分节点 `lambdaRef` 变换到 K 上为

```

1 % quadrature points in the x-y coordinate
2 pxy = lambdaRef(p,1)*node(elem(:,1),:) ...
3     + lambdaRef(p,2)*node(elem(:,2),:) ...
4     + lambdaRef(p,3)*node(elem(:,3),:);

```

由此可计算相应的真实节点值

```

1 elemu = zeros(NT,3*ng);
2 for p = 1:3*ng
3     % quadrature points in the x-y coordinate
4     pxy = lambdaRef(p,1)*node(elem(:,1),:) ...
5         + lambdaRef(p,2)*node(elem(:,2),:) ...
6         + lambdaRef(p,3)*node(elem(:,3),:);
7     elemu(:,p) = uexact(pxy);
8 end

```

当然, 我们不需要 `pxy` 的信息.

`elemuh` 每行对应一个单元, 前 n_g 列对应第 1 条边, 中间 n_g 列对应第 2 条边, 最后 n_g 列对应第 3 条边. 边界边由于只出现一次, 红和黑对应的插值必然有一个全为零. 这恰好符合跳量的计算. 将所有节点值根据编号排成一列, 记为 `uhI`, 则有

```

1 uhI = zeros(2*NE*ng,1);
2 uhI(elem2dofe) = elemuh;

```

12.1.3 跳量与跳量积分的计算

现在我们按单元计算跳量, 它与 `elemuh` 的尺寸一样, 每行对应一个单元, 每列对应一个积分点处的跳跃.

- 以下将前面给出的 `elemuh` 记为 `elemuhP` (`P` 表示 Plus), 表示边左侧的值 (只是相对而言). 为此, 我们只要给出右侧的值 `elemuhM` (`M` 表示 Minus), 两者相减就可获得跳量. 将前面给出的 `elem2dofe` 记为 `elem2dofP`, 只要对应 `elem2dofP` 给出 `elem2dofM`, 则有

```

1 elemuhM = uhI(elem2dofM);

```

- 对给定单元的一条边, 设它在 `elem2dofP` 中的积分节点编号为 i_1, \dots, i_{n_g} , 相邻一侧的编号记为 i'_1, \dots, i'_{n_g} . 易知 $|i_k - i'_k| = n_g \cdot N_E$, 即对应编号相差 $n_g \cdot N_E$. 黑色边对应的序号从 1 到 $n_g \cdot N_E$. 这样, 只要把 `elem2dofP` 中大于 $n_g \cdot N_E$ 的序号减去 $n_g \cdot N_E$, 小于或等于 $n_g \cdot N_E$ 的序号加上 $n_g \cdot N_E$, 所得即为 `elem2dofM`.

```

1 index = (elem2dofP > ng*NE);
2 elem2dofM = elem2dofP + (-ng*NE)*index + ng*NE*(-index);

```

对给定函数的插值 u_h , 经检查, 内部边的跳量都为零或接近零.

有了按单元边存储的跳量 (加上了平方)

```

1 elem2Jumpval = (elemuhP - elemuhM).^2;

```

我们就可计算跳量积分

$$\eta_K^2 := \sum_{e \subset \partial K} c_e \| [u_h] \|_{0,e}^2 \approx \sum_{e \subset \partial K} c_e h_e [w_1 [u_h]^2(p_1) + \dots + w_{n_g} [u_h]^2(p_{n_g})].$$

- `elemJumpval` 的前 n_g 列对应第 1 条边, 中间 n_g 列对应第 2 条边, 最后 n_g 列对应第 3 条边. 假设 $c_e = 1$ (h_e 的处理同 c_e), 则第 i 条边对应的 h_e 列向量为

```

1 hei = he(elem2edge(:, i));

```

从而积分为

```

1 elemJumpei = hei.*(elem2Jumpval(:, (1:ng)+(i-1)*ng)*weight1d(:));

```

循环求和即得所有单元对应的 η_K^2 .

12.2 Poisson 方程的自适应有限元方法

12.2.1 后验误差估计

考虑 Poisson 方程的 Dirichlet 问题

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega, \end{cases} \quad (12.1)$$

为了方便, 本文只考虑 $g = 0$ 的情形. 变分问题为: 找 $u \in V = H_0^1(\Omega)$ 使得

$$a(u, v) = \ell(v), \quad v \in V,$$

式中,

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v dx, \quad \ell(v) = \int_{\Omega} f v dx.$$

线性有限元方法为: 找 $u_h \in V_h$ 使得

$$a(u_h, v) = \ell(v), \quad v \in V_h, \quad (12.2)$$

这里, 有限元空间

$$V_h = \{v \in H^1(\Omega) \cap C(\bar{\Omega}) : v|_K \in \mathbb{P}_1(K) \quad \forall K \in \mathcal{T}_h, \quad v|_{\partial\Omega} = 0\},$$

而 \mathcal{T}_h 是满足 shape regularity 条件的三角剖分.

对协调有限元问题 (12.2), 利用 Ceá 引理, 误差估计转化为逼近算子 (例如插值算子) 的误差估计, 通常可建立如下的先验估计

$$\|u - u_h\|_{0,K} + h_K |u - u_h|_{1,K} \lesssim h_K^{s+1} |u|_{s+1,K}, \quad s \geq 0.$$

这里, 右端依赖于未知函数 u 的先验信息 (如 H^2 正则性), 因而称为先验估计.

后验误差估计是建立如下类型的估计

$$\|u - u_h\| \lesssim \eta(u_h),$$

式中, $\|\cdot\|$ 是某种范数或半范数, 用以度量误差, η 是与未知函数 u 无关的上界, 且与数值解 u_h 相关, 即 $\eta = \eta(u_h)$. 对问题 (12.2), 我们可建立如下的估计

$$|u - u_h|_1 \lesssim \eta(u_h),$$

式中,

$$\eta = \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{1/2}, \quad \eta_K^2 = h_K^2 \|f + \Delta u_h\|_{0,K}^2 + \sum_{e \subset \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2.$$

这里, $f + \Delta u_h$ 显然是数值解关于方程的残差, 注意, 对线性元 $\Delta u_h = 0$, 而

$$[\partial_{n_e} u_h] = \partial_{n_e} u_h|_{K_1} - \partial_{n_e} u_h|_{K_2}$$

为 $\partial_{n_e} u_h$ 跨单元时的跳量, n_e 是对应 K_1 的外法向量.

- 对给定的网格剖分 \mathcal{T}_h , 数值解在每个单元上计算的精度是不同的. 有些单元可能计算得好, 而有些单元可能计算得差, 这与区域以及解的正则性等都有关系. 例如, 对 L 型区域的角点附近, 通常网格剖分要细密一点, 否则这部分计算得就相对差一点. 再比如, 若精确解在某点附近有奇异性, 则这部分网格也要密一点.
- η_K 称为局部误差指示子 (local error indicator), 它的大小可用来度量单元 K 上的计算好坏. 当 η_K 较大时, 我们可以对该单元进行局部加密.

为了方便, 对 $\mathcal{M} \subset \mathcal{T}_h$, 定义

$$\eta(u_h, \mathcal{M}) = \left(\sum_{K \in \mathcal{M}} \eta_K^2 \right)^{1/2}.$$

显然, 有 $\eta(u_h) = \eta(u_h, \mathcal{T}_h)$.

12.2.2 加密或标记准则

现在给出一些局部加密的准则, 常用的有以下三种.

1. **The error equidistribution strategy** (误差均分策略)

给定 $\theta > 1$, 标记所有的单元 K^* , 满足

$$\eta_{K^*} \geq \theta \frac{\text{TOL}}{\sqrt{NT}},$$

式中, TOL 相当于迭代算法中停止的容许值, NT 是单元的总个数.

2. **The maximum marking strategy** (Babuska-Rheinboldt strategy)

标记所有的单元 K^* , 使得

$$\eta_{K^*} \geq \theta \max_{K \in \mathcal{T}_h} \eta_K, \quad \theta \in (0, 1).$$

3. **The Dörfler bulk strategy**

标记所有的单元 $K^* \subset \mathcal{M}_h$, 使得

$$\eta^2(u_h, \mathcal{M}_h) \geq \theta \eta^2, \quad \theta \in (0, 1),$$

这里, \mathcal{M}_h 是 \mathcal{T}_h 的子集. 注意, 通常将 η_K^2 从大到小排列, 找到最少的前若干项单元作为 \mathcal{M}_h .

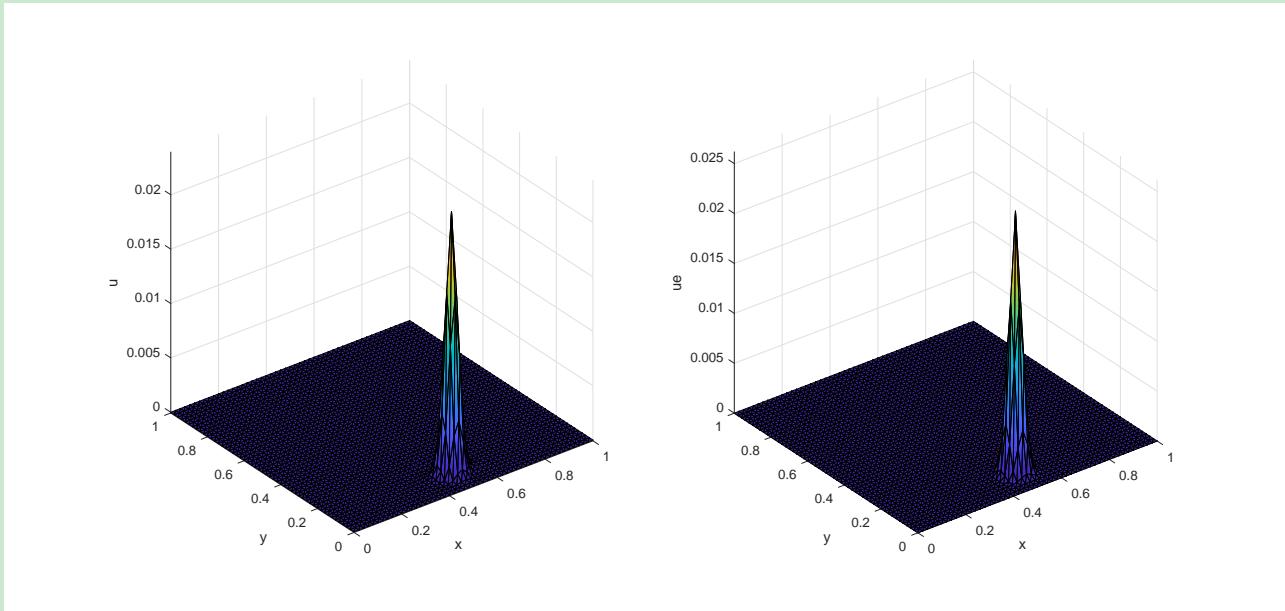
12.2.3 Poisson 方程的自适应有限元程序

为了方便操作, 我们先给出一个初始网格上的问题. 考虑 Dirichlet 问题 (12.1), 精确解设为

$$u(x, y) = xy(1-x)(1-y)\exp\left(-1000((x-0.5)^2 + (y-0.117)^2)\right),$$

区域 $\Omega = (0, 1)^2$, 此时 $g = 0$. 这个解衰减得很快, 它在 $(0.5, 0.117)$ 处的值要明显大于其他处的, 为此真正计算过程中, 只需要该点附近的网格细密就可 (不妨称为奇点).

Poisson 方程的计算程序前面已经给出, 横纵划分 50 等份的结果如下



我们先直接给出自适应方法的主程序，后面再逐一给出每一步的处理过程。主程序如下

```

1 clc;clear;close all;
2
3 %% Parameters
4 maxN = 1e4;      theta = 0.4;      maxIt = 100;
5 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
6
7 %% Generate an initial mesh
8 Nx = 4; Ny = 4; h1 = 1/Nx; h2 = 1/Ny;
9 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
10
11 %% Get the PDE data
12 pde = Poissondata_afem();
13
14 %% Adaptive Finite Element Method
15 for k = 1:maxIt
16     % Step 1: SOLVE
17     bdStruct = setboundary(node,elem);
18     u = Poisson(node,elem,pde,bdStruct);
19     figure(1);
20     showresult(node,elem,pde.uexact,u);
21
22     % Step 2: ESTIMATE
23     eta = Poisson_indicator(node,elem,u,pde);
24
25     % Step 3: MARK
26     elemMarked = mark(elem,eta,theta);
27
28     % Step 4: REFINE
29     [node,elem] = bisect(node,elem,elemMarked);

```

```

30
31     if (size(node,1)>maxN) || (k==maxIt)
32         bdStruct = setboundary(node,elem);
33         u = Poisson(node,elem,pde,bdStruct);
34         setp = k
35         break;
36     end
37
38 end

```

该程序中涉及三个重要模块.

- Step 2 的误差指示子
- Step 3 的标记算法
- Step 4 的加密算法

网格剖分如下

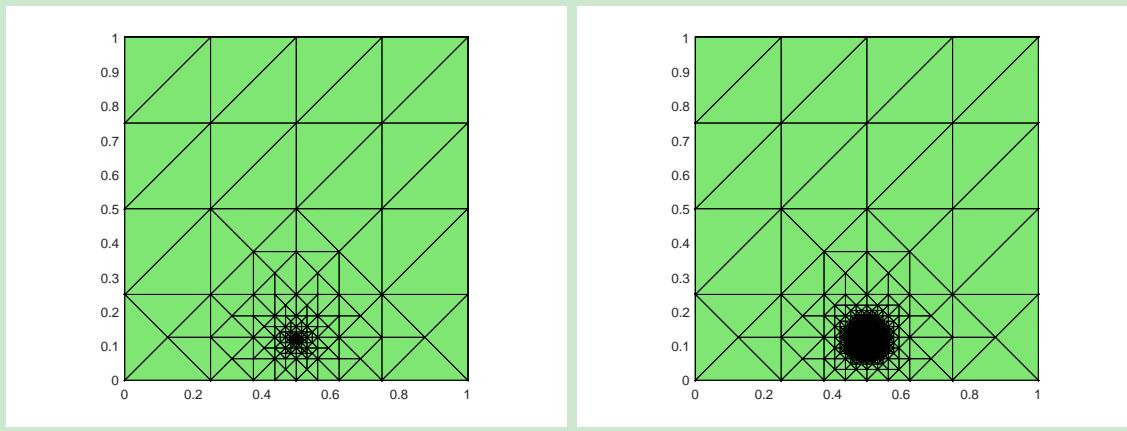


图 12.2. AFEM 的网格剖分

可以看到, 网格加密基本上在奇点附近.

迭代 30 次的数值解与精确解如下.

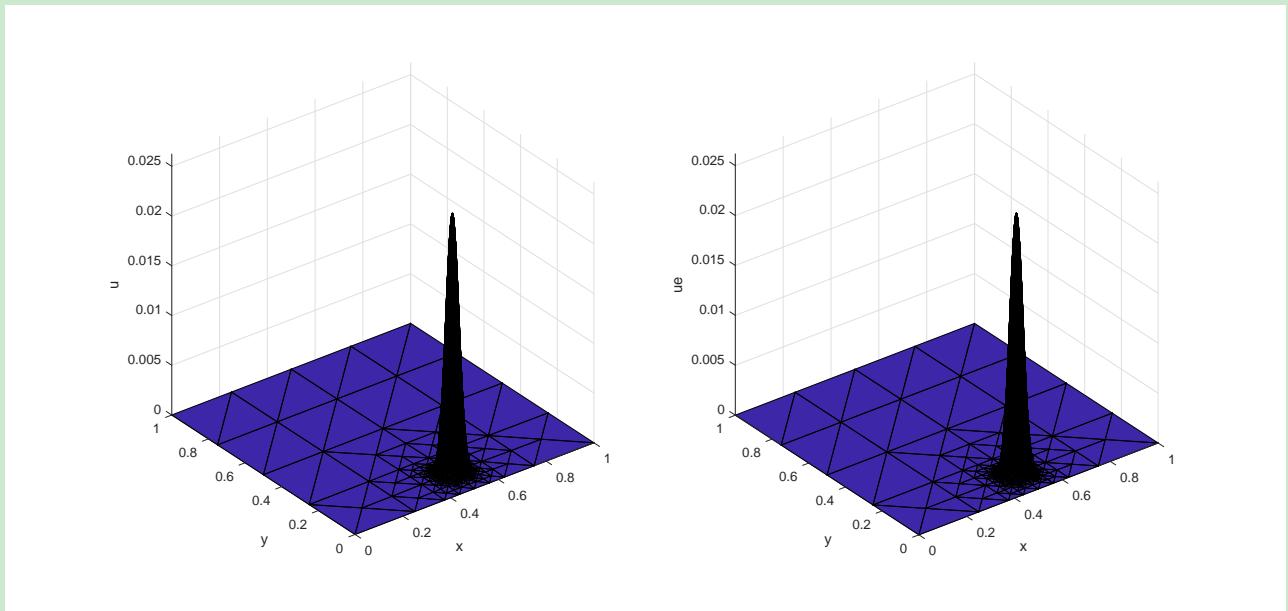


图 12.3. AFEM 的数值解与精确解

可以看到, 计算精度要比同时划分 50 份的高.

12.2.4 误差指示子的计算

现在考虑误差指示子

$$\eta_K^2 = h_K^2 \|f\|_{0,K}^2 + \sum_{e \subset \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2$$

的计算, 这里右端第一项对应残差, 记为 elemRes; 第二项对应跳量, 记为 elemJump. 注意它们都是 NT 个分量的向量, 每个位置对应一个单元.

残差的计算

右端第一部分的处理是熟悉的, 用 Gauss 求积公式计算.

```

1 %% elementwise residuals
2 [lambda,weight] = quadpts(3);
3 NT = size(elem,1);
4 elemRes = zeros(NT,1);
5 for p = 1:length(weight)
6     % quadrature points in the x-y coordinate
7     pxy = lambda(p,1)*node(elem(:,1),:) ...
8         + lambda(p,2)*node(elem(:,2),:) ...
9         + lambda(p,3)*node(elem(:,3),:);
10    fxy = pde.f(pxy);
11    elemRes = elemRes + weight(p)*fxy.^2;
12 end
13 elemRes = diameter.^2.*area.*elemRes;

```

注 12.1 iFEM 中定义 $h_K = |K|^{1/2}$, 这由 shape regularity 保证. 这样, $h_K^2 = |K|$, 再考虑到积分公式前面的 $|K|$, iFEM 中会出现 $h_K^2 \cdot |K| = |K|^2$, 这就是那里的函数 estimateresidual.m 计算时给出如下语句的原因

```
elemResidual = elemResidual.* (area.^2);
```

边界跳量的计算方法一

现在计算第二项

$$\sum_{e \subset \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2,$$

其中 $\partial_{n_e} u_h = \nabla u_h \cdot n_e$. 注意到一次函数求导后为常函数, 这里的特例可直接按照边集合的方式进行计算.

- 给定左侧单元 K_1 及其边 e , 设右侧单元为 K_2 . 再设 K_1 的边界外法向量为 n_e , 则跳量为

$$[\partial_{n_e} u_h] = [\nabla u_h \cdot n_e] = \nabla u_h \cdot n_e|_{K_1} - \nabla u_h \cdot n_e|_{K_2} = (\nabla u_h|_{K_1} - \nabla u_h|_{K_2}) \cdot n_e.$$

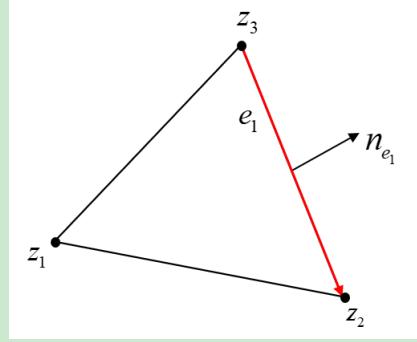
- 注意, 对左右单元跳量是一致的, 从而 $h_e \|[\partial_{n_e} u_h]\|_{0,e}^2$ 也是一致的. 注意到 ∇u_h 是常数, 我们有

$$\begin{aligned} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2 &= h_e \int_e [\nabla u_h \cdot n_e]^2 ds = h_e \int_e [\nabla u_h \cdot n_e]^2 ds \\ &= h_e^2 [\nabla u_h \cdot n_e]^2 = [\nabla u_h \cdot h_e n_e]^2 \end{aligned}$$

因结果是正的, 我们不必关心 n_e 的方向, 只要取绝对值即可.

- 辅助数据结构中:
 - `edge` 给出了一维边的端点标记, 行索引对应自然序号;
 - `elem2edge` 给出了按单元存储的三条边的自然序号;
 - `edge2elem`, 它给出了边 e 的左右单元.
- 为了避免重复计算, 我们将把所有的一维边 e 的结果计算出来, 然后再给出每个单元的结果.

先考虑 $h_e n_e$ 的计算, 我们按照一维边集合 `edge` 计算, 结果保存为 `ne`.



如图, 三角形 z_i 的对边为 e_i , 相应的外法向量为 n_{e_i} . 可以看到, $h_{e_1}n_{e_1}$ 可由边向量 $e_1 = \overrightarrow{z_3z_2}$ 逆时针旋转 90° 获得. 易知, 向量 $\alpha = (a, b)$ 旋转 90° 后为 $(-b, a)$. 如下获得所有一维边的 $h_e n_e$ (忽略方向):

```

1 % edge vectors
2 e = node(edge(:,2),:)-node(edge(:,1),:);
3 % scaled norm vectors he*ne
4 ne = [-e(:,2),e(:,1)]; % stored in rows

```

把 h_e 和 n_e 放在一起, 避免了求边长.

再考虑 ∇u_h 的计算. 注意, 我们已经在初始网格 \mathcal{T}_h 上获得了 u_h (即每个节点处的值, 为了方便, 第 i 个节点的值记为 u_i), 据此可计算 ∇u_h . 设三角形 K 的三个顶点为 z_1, z_2, z_3 , 相应的数值解为 u_1, u_2, u_3 , 则在该单元上有

$$u_h = u_1\lambda_1 + u_2\lambda_2 + u_3\lambda_3,$$

其中, $\lambda_1, \lambda_2, \lambda_3$ 是相应的节点基. 于是,

$$\nabla u_h = u_1\nabla\lambda_1 + u_2\nabla\lambda_2 + u_3\nabla\lambda_3,$$

为此需要计算 $\nabla\lambda_i$. 可以证明,

$$\begin{aligned} \frac{\partial\lambda_1}{\partial x} &= \frac{1}{2|K|}(y_2 - y_3), & \frac{\partial\lambda_1}{\partial y} &= \frac{1}{2|K|}(x_3 - x_2), \\ \frac{\partial\lambda_2}{\partial x} &= \frac{1}{2|K|}(y_3 - y_1), & \frac{\partial\lambda_2}{\partial y} &= \frac{1}{2|K|}(x_1 - x_3), \\ \lambda_3 &= 1 - \lambda_1 - \lambda_2. \end{aligned}$$

左右单元的如下计算

```

1 % information of left and right elements
2 k1 = edge2elem(:,1); k2 = edge2elem(:,2);
3 index1 = elem(k1,:); index2 = elem(k2,:);
4 zL1 = node(index1(:,1),:); zR1 = node(index2(:,1),:);
5 zL2 = node(index1(:,2),:); zR2 = node(index2(:,2),:);

```

```

6 zL3 = node(index1(:,3),:);    zR3 = node(index2(:,3),:);
7
8 % grad of nodal basis functions
9 gradL1 = [zL2(:,2)-zL3(:,2), zL3(:,1)-zL2(:,1)]; % stored in rows
10 gradL2 = [zL3(:,2)-zL1(:,2), zL1(:,1)-zL3(:,1)];
11 gradR1 = [zR2(:,2)-zR3(:,2), zR3(:,1)-zR2(:,1)];
12 gradR2 = [zR3(:,2)-zR1(:,2), zR1(:,1)-zR3(:,1)];
13
14 gradL1 = gradL1./(2*area(k1)); gradR1 = gradR1./(2*area(k2));
15 gradL2 = gradL2./(2*area(k1)); gradR2 = gradR2./(2*area(k2));
16 gradL3 = -(gradL1+gradL2);      gradR3 = -(gradR1+gradR2);

```

这里, 梯度向量按行排列坐标, 每行对应一个边.

据此, 可如下计算每条边 e 对应的 $[\nabla u_h]$

```

1 % grad of uh
2 gradLu = u(index1(:,1)).*gradL1 + u(index1(:,2)).*gradL2 + u(index1(:,3)).*gradL3;
3 gradRu = u(index2(:,1)).*gradR1 + u(index2(:,2)).*gradR2 + u(index2(:,3)).*gradR3;
4 % jump of gradu
5 Jumpu = gradLu-gradRu;
6 Jumpu(k1==k2,:) = gradLu(k1==k2,:);

```

这里, $k1==k2$ 表示边界上的, 值为本身 (对齐次 Dirichlet 边界部分, 因检验函数在边界上为零, 跳量部分可以去掉, 例如 iFEM. 本文保留).

由此可得到边集合对应的 $h_e \|\partial_{n_e} u_h\|_{0,e}^2$ 的结果, 记为 edgeJump

```

1 % edgeJump
2 edgeJump = dot(Jumpu', nedge').^2; edgeJump = edgeJump';

```

注意, 对两个相同的矩阵, `dot` 的作用时对应列进行运算. 因为我们要把每行对应的向量点乘, 故进行转置再 `dot`.

知道每条边的跳量后, 我们可以利用辅助数据结构 elem2edge 获得单元的跳量 $\sum_{e \in \partial K} h_e \|\partial_{n_e} u_h\|_{0,e}^2$.

```

1 % elemJump
2 elemJump = sum(edgeJump(elem2edge), 2);

```

`edgeJump` 是一个列向量, 行索引对应边的自然序号. `elem2edge` 存储单元边的自然序号, 它是 $NT \times 3$ 的矩阵, 第 1 列对应每个单元的第 1 条边, 依此类推. `edgeJump(elem2edge)` 是与 `elem2edge` 相同维数的矩阵, 每个位置给出对应边的跳量. 显然按行求和就得到每个单元的跳量.

综上, 误差指示子为

```

1 % ----- Local error indicator -----
2 eta = (abs(elemRes) + elemJump).^(1/2);

```

注 12.2 quadpts(3) 给出的权重含有负值, 这一点非常奇怪. 为此, 上面计算的 elemRes 加了绝对值, 否则会出现对负数开平方.

边界跳量的计算方法二

也可以使用本章开始提供的设计思路进行计算. 易知, 有

$$h_e \|[\partial_{n_e} u_h]\|_{0,e}^2 = h_e \int_e ([\partial_x u_h] n_x + [\partial_y u_h] n_y)^2 ds.$$

- 边界积分的连通性矩阵 elem2dofP 和 elem2dofM 的获取过程不变.
- 对局部基函数 ϕ_1, ϕ_2, ϕ_3 , 在固定单元上, 它们的导数为常数, 从而每个积分节点处的值相同. 这样, 只要存储所有单元的

$$\text{basex} = (\phi_{1,x}, \phi_{2,x}, \phi_{3,x}), \quad \text{basey} = (\phi_{1,y}, \phi_{2,y}, \phi_{3,y}).$$

```

1 Dlambda = gradbasis(node, elem);
2 grad1 = Dlambda(:, :, 1); grad2 = Dlambda(:, :, 2); grad3 = Dlambda(:, :, 3);
3 basex = [grad1(:, 1), grad2(:, 1), grad3(:, 1)]; % Dxphi1, Dxphi2, Dxphi3
4 basey = [grad1(:, 2), grad2(:, 2), grad3(:, 2)];

```

- x, y 的偏导数插值如下计算

```

1 Ndof = 3;
2 for p = 1:3*ng
3     % interpolation at the p-th quadrature point
4     for i = 1:Ndof
5         elemuhxP(:, p) = elemuhxP(:, p) + uh(elem2dof(:, i)).*basex(:, i);
6         elemuhyP(:, p) = elemuhyP(:, p) + uh(elem2dof(:, i)).*basey(:, i);
7     end
8 end
9 uhxI = zeros(2*NE*ng, 1);      uhyI = zeros(2*NE*ng, 1);
10 uhxI(elem2dofP) = elemuhxP;   uhyI(elem2dofP) = elemuhyP;
11 elemuhxM = uhxI(elem2dofM);   elemuhyM = uhyI(elem2dofM);

```

这里的 elemuhxM 是对应 elemuhxP 的右侧值.

- 于是, 跳量的积分为

```

1 %% elementwise jump at quadrature points
2 elem2Jumpx = elemuhxP - elemuhxM;
3 elem2Jumpy = elemuhyP - elemuhyM;
4 elemJump = zeros(NT, 1);
5 for i = 1:3
6     hei = he(elem2edge(:, i));
7     neix = ne(elem2edge(:, i), 1); neiy = ne(elem2edge(:, i), 2);

```

```

8      Jumpnx = elem2Jumpx(:,(1:ng)+(i-1)*ng).*repmat(neix,1,ng);
9      Jumpny = elem2Jumpy(:,(1:ng)+(i-1)*ng).*repmat(neiy,1,ng);
10     Jumpn = (Jumpnx+Jumpny).^2;
11     elemJump = elemJump + hei.^2.* (Jumpn*weightid(:));
12 end

```

注 12.3 也可以恢复边界的逆时针定向, 如下更改 sgnelem:

```

1 % sign of elementwise edges
2 [~, i2] = unique(totalEdge(end:-1:1,:),'rows');
3 i2 = size(totalEdge,1)+1-i2; % last occurrence
4 bdEdgeIdx = (i1==i2);
5 sgnelem = sign([elem(:,3)-elem(:,2), ...
6                 elem(:,1)-elem(:,3), ...
7                 elem(:,2)-elem(:,1)]);
8 E = false(NE,1); E(bdEdgeIdx) = 1; sgnbd = E(elem2edge);
9 sgnelem(sgnbd) = 1;

```

计算结果一致.

indicator 函数

第二种方法的过程编写为如下函数.

```

1 function eta = Poisson_indicator(node, elem, uh, pde)
2 % This function returns the local error indicator of Poisson equation with
3 % homogeneous Dirichlet boundary condition in 2-D.
4
5 % Copyright (C) Terence Yu.
6
7 %% preparation for the computation
8 % auxiliary data
9 aux = auxgeometry(node, elem);
10 area = aux.area; diameter = aux.diameter;
11 auxT = auxstructure(node, elem);
12 elem2edge = auxT.elem2edge;
13 edge = auxT.edge;
14 % he, ne
15 e = node(edge(:,2),:)-node(edge(:,1),:);
16 he = sqrt(sum(e.^2, 2));
17 ne = [-e(:,2), e(:,1)]; % stored in rows
18 ne = ne./repmat(he, 1, 2);
19 % number
20 NT = size(elem, 1); NE = size(edge, 1);
21 % elem2dof
22 elem2dof = elem;
23 % sign of elementwise edges
24 sgnelem = sign([elem(:,3)-elem(:,2), elem(:,1)-elem(:,3), elem(:,2)-elem(:,1)]);
25 % Dlambd

```

```

26 Dlambda = gradbasis(node,elem);
27 grad1 = Dlambda(:,:,1);
28 grad2 = Dlambda(:,:,2);
29 grad3 = Dlambda(:,:,3);
30
31 %% elementwise residuals
32 [lambda,weight] = quadpts(3);
33 elemRes = zeros(NT,1);
34 for p = 1:length(weight)
35     % quadrature points in the x-y coordinate
36     pxy = lambda(p,1)*node(elem(:,1),:) ...
37         + lambda(p,2)*node(elem(:,2),:) ...
38         + lambda(p,3)*node(elem(:,3),:);
39     fxy = pde.f(pxy);
40     elemRes = elemRes + weight(p)*fxy.^2;
41 end
42 elemRes = diameter.^2.*area.*elemRes;
43
44 %% connectivity list of jump integral
45 [~,weight1d] = quadpts1(4); ng = length(weight1d);
46 elem2dofP = zeros(NT,3*ng); % P: plus +
47 %elem2dofM = zeros(NT,3*ng); % M: minus -
48 for i = 1:3 % i-th side
49     ei = elem2edge(:,i); sgni = sgnelem(:,i);
50     idP = repmat(1:ng, NT,1);
51     idP(sgni<0,:) = repmat((ng:-1:1)+NE*ng, sum(sgni<0), 1);
52     elem2dofP(:,(1:ng)+(i-1)*ng) = idP + repmat((ei-1)*ng,1,ng);
53 end
54 index = (elem2dofP>ng*NE);
55 elem2dofM = elem2dofP + (-ng*NE)*index + ng*NE*(-index);
56
57 %% elementwise interpolant at quadrature points
58 elemuhxP = zeros(NT,3*ng);
59 elemuhyP = zeros(NT,3*ng);
60 baseX = [grad1(:,1),grad2(:,1),grad3(:,1)]; % Dxphi1,Dxphi2,Dxphi3
61 baseY = [grad1(:,2),grad2(:,2),grad3(:,2)];
62 Ndof = 3;
63 for p = 1:3*ng
64     % interpolation at the p-th quadrature point
65     for i = 1:Ndof
66         elemuhxP(:,p) = elemuhxP(:,p) + uh(elem2dof(:,i)).*baseX(:,i);
67         elemuhyP(:,p) = elemuhyP(:,p) + uh(elem2dof(:,i)).*baseY(:,i);
68     end
69 end
70 uhxI = zeros(NT*3*ng,1); uhyI = zeros(NT*3*ng,1);
71 uhxI(elem2dofP) = elemuhxP; uhyI(elem2dofP) = elemuhyP;
72 elemuhxM = uhxI(elem2dofM); elemuhyM = uhyI(elem2dofM);
73
74 %% elementwise jump at quadrature points
75 elem2JumpX = elemuhxP - elemuhxM;

```

```

76 elem2Jump = elemuhyp - elemuhym;
77 elemJump = zeros(NT,1);
78 for i = 1:3
79     hei = he(elem2edge(:,i));
80     neix = ne(elem2edge(:,i),1); neiy = ne(elem2edge(:,i),2);
81     Jumpnx = elem2Jumpx(:,(1:ng)+(i-1)*ng).*repmat(neix,1,ng);
82     Jumpny = elem2Jumpy(:,(1:ng)+(i-1)*ng).*repmat(neiy,1,ng);
83     Jumpn = (Jumpnx+Jumpny).^2;
84     elemJump = elemJump + hei.^2.* (Jumpn*weight1d(:));
85 end
86
87 %% Local error indicator
88 eta = (abs(elemRes) + elemJump).^(1/2);

```

经比较发现, 两种方法给出的结果完全相同.

12.3 标记算法的实现

以下只考虑后两种标记算法, 且假设 η_K 都已获得.

1. The maximum marking strategy

标记所有的单元 K^* , 使得

$$\eta_{K^*} \geq \theta \max_{K \in \mathcal{T}_h} \eta_K, \quad \theta \in (0, 1).$$

程序如下

```

1 NT = size(elem,1); isMark = false(NT,1);
2 isMark(eta>theta*max(eta)) = 1;
3 elemMarked = find(isMark==true);

```

2. The Dörfler bulk strategy

标记所有的单元 $K^* \subset \mathcal{M}_h$, 使得

$$\eta^2(u_h, \mathcal{M}_h) \geq \theta \eta^2, \quad \theta \in (0, 1).$$

将单元按 η_K^2 从大到小排列, 并命名为 K_1, K_2, \dots . 定义

$$s(n) = \sum_{i=1}^n \eta_{K_i}^2,$$

则要找到最小的 n^* 使得,

$$s(n^*) \geq \theta \eta^2 = \theta s(NT).$$

显然 n_* 存在, 且至多为 NT . 我们可转而找最大的 n_* 使得

$$s(n_*) < \theta \eta^2 = \theta s(NT). \tag{12.3}$$

这样, K_1, \dots, K_{n_*} 都是需要标记的单元. 自然 $n_* < NT$, 故 $n^* = n_* + 1$. 为了方便, 我们只标记 K_1, \dots, K_{n_*} . 注意, 若没有满足条件 (12.3) 的 n_* , 则表明 K_1 就是需要的. 程序如下

```

1 [sortedEta,id] = sort(eta.^2,'descend');
2 x = cumsum(sortedEta);
3 isMark(id(x < theta*x(end))) = 1;
4 isMark(id(1)) = 1;
5 elemMarked = find(isMark==true);

```

这里, `cumsum` 是累加函数.

综上, 标记程序如下

CODE 12.1. mark.m

```

1 function elemMarked = mark(elem,eta,theta,method)
2 %Mark mark element.
3
4 NT = size(elem,1); isMark = false(NT,1);
5 if ~exist('method','var'), method = 'Dorfler'; end
6 % default marking is Dofler bulk strategy
7 switch strcmpi(method,'max')
8     case true
9         isMark(eta>theta*max(eta))=1;
10    case false
11        [sortedEta,id] = sort(eta.^2,'descend');
12        x = cumsum(sortedEta);
13        isMark(id(x < theta*x(end))) = 1;
14        isMark(id(1)) = 1;
15    end
16 elemMarked = find(isMark==true);

```

这里, `strcmpi(method, 'max')` 是进行字符串比较且不区分大小写, 因此只要 `method` 是 `max` (不区分大小写), 则选择第一种方法, 其他任何情况都是 Dörfler.

注 12.4 对 maximum marking strategy, θ 越大, 标记的单元越少; Dörfler bulk strategy 恰恰相反.

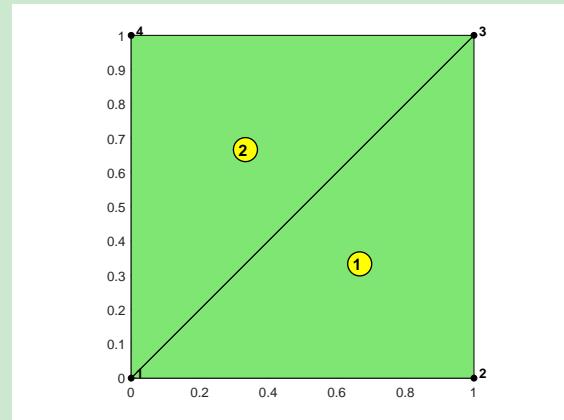
12.4 最新点二分

现在介绍一种局部加密算法, 称为 Newest-node bisection, 不妨翻译为最新点二分. 加密时要注意两点:

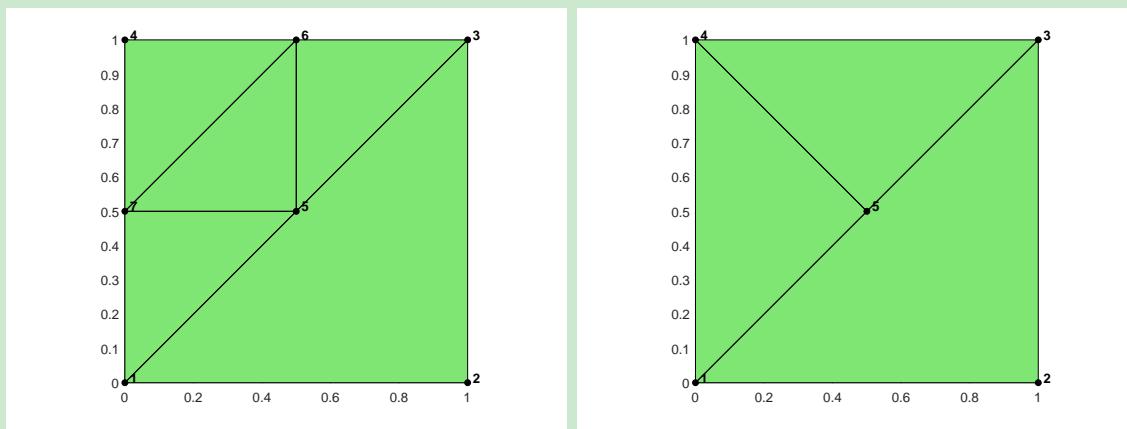
- 保持 shape regularity: 等价于最小角条件;
- 保持协调: 无悬挂点.

12.4.1 局部加密方式

标记好单元后, 我们就要对它们进行局部加密. 以如下的初始网格为例进行说明, 且考虑对单元 2 进行加密.



通常有两种局部加密方式, 如下图所示



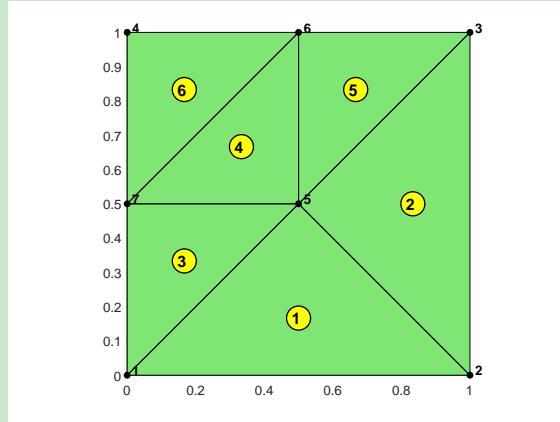
(a) regular refinement

(b) bisection refinement

图 12.4. 两种加密方式

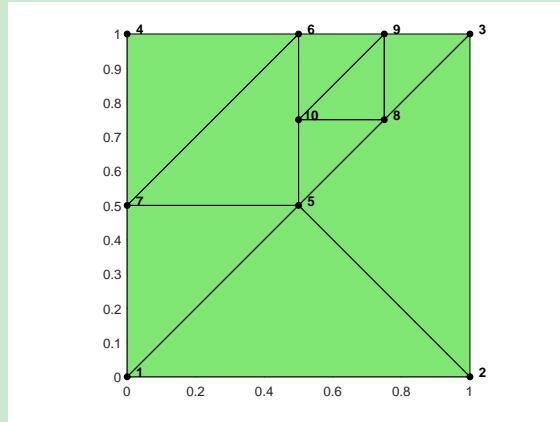
- 正规加密 (regular refinement): 连接三角形各边的中点.
- 二分加密 (bisection refinement): 连接三角形某个顶点与其对边的中点. 称该顶点为三角形的 peak, 对边为 base.

可以看到, 这两种加密均导致非协调剖分, 即含有悬挂点 5. 为了解决悬挂点 5, 对正规加密的图 12.4 (a), 可连接 2-5, 即对单元 ① 进行二分, 如下图

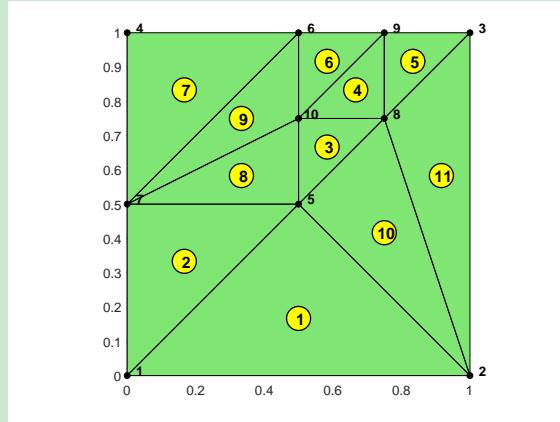


通常称这种处理悬挂点的方法为: the bisection of a triangle as a green refinement. 需要注意的是, 处理正规加密悬挂点的 green refinement 可能导致三角形的退化. 这里退化的意思是剖分不再满足 shape regularity 条件, 因为此时三角形的角可能变得很小.

例如继续对单元 ⑤ 进行正规加密, 得



此时会出现悬挂点 8 和 10, 从而要连接 7-10 和 2-8, 得 green refinement



继续对图中的单元 ⑤ 进行加密的话, 单元 ⑪ 又被二分, 从而三角形的角会越来越小.

注 12.5 正规加密可以说是一种自然加密, 因为它在加密过程中产生的子三角形与大三角形是相似的, 从而自然地继承了 shape regularity. 本文考虑二分加密.

12.4.2 最新点二分的简单说明

为了简明, 我们用一个例子来说明最新点二分.

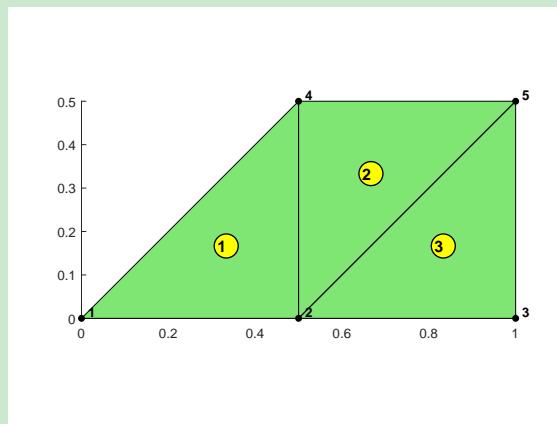


图 12.5. 初始剖分

peak 与 base

如图, 设单元①和③是标记单元, 即 `elemMarked = [1; 3]`. 对一个三角形, 二分是连接某个顶点与对边的中点, 称该顶点为 peak, 对边为 base.

peak 可以是任意一个, 注意到三角形是按顶点逆时针顺序记录的, 循环置换仍表示同一个三角形, 即 `elem(t, [1 2 3])`, `elem(t, [2 3 1])` 和 `elem(t, [3 1 2])` 都表示单元三角形 t . 为此, 我们规定: 第 1 个顶点为 peak, 即 `elem(t, 1)` 是 peak. 根据数据结构的规定, 第 1 个顶点的对边为第 1 条边, 它就是相应的 base, 即为 `elem(t, [2 3])`. 现在假设, 单元①和③的 peak 分别为 1, 3.

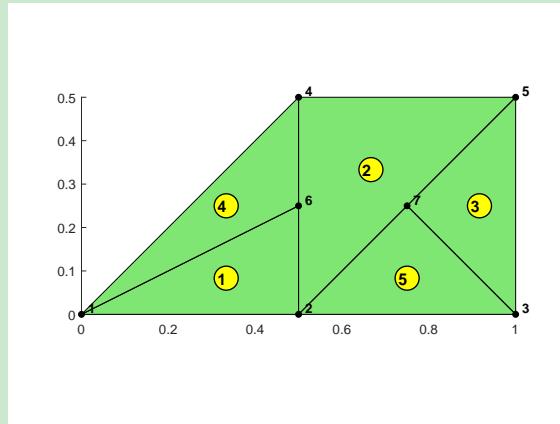
辅助数据结构中, `elem2edge` 按单元存储边的自然序号, 从而 base 可用 `elem2edge(t, 1)` 获得自然序号, 这样, 标记单元的 base 在边集合中的自然序号为

```
base = elem2edge(elemMarked, 1);
```

在一个三角形中 peak 与 base 是一一对应的, 只要记录 base 即可.

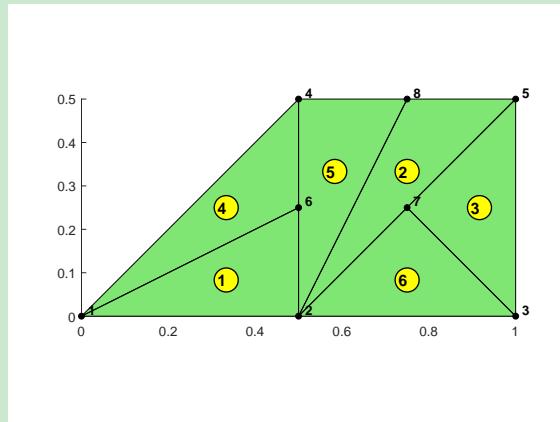
标记单元的扩充

现在开始对标记单元进行二分, 结果为



对每个单元, 规定: 二分的左侧子三角形保留原编号, 右侧子三角形从 $NT+1$ 开始按顺序编号. 为了处理悬挂点, 必须对 base 的相邻单元 ② 进行剖分. 这表明, base 的左右单元都要标记. 而单元与 base 是对应的, 为此只需要记录 base 即可.

最新点二分是把 base 的中点作为子三角形的 peak. 一个重要观察是, 单元 ② 的任一顶点作为 peak 都可解决悬挂点问题. 若 4 是 peak, 则连接 4-7 后悬挂点 7 消失. 根据规定, 7 是子三角形 7-4-2 的 peak. 这样, 单元 1-2-4 与子三角形 7-4-2 有公共的 base, 它们分别连接 peak 和 base, 悬挂点 6 也消失. 若 5 是 peak, 则与上面的情形类似. 若 2 是悬挂点, 连接对边后有



此时又划归到有公共 base 的情形, 即子三角形 8-4-2 与单元 1-2-4 有公共 base 2-4, 而子三角形 8-5-2 与单元 3-5-2 有公共 base 5-2.

注 12.6 当解决悬挂点后, 我们不再进行二分. 这保证了 base 都是原来的 edge, 即没有新的边成为 base. 称其为 “base-edge” 性质.

注 12.7 “base-edge” 性质使得我们可方便地进行二次加密, 以去除悬挂点. 例如, 在上图中 base 只有 2-4, 5-4 和 5-2. 对单元 ①, 按规定, 尽管 6 是 peak, 但其对边不在 base 集合中, 从而不会发生二分. 单元 ⑤ 则不同, 它的 peak 为 8, 对边 4-2 在 base 集合中, 因而会

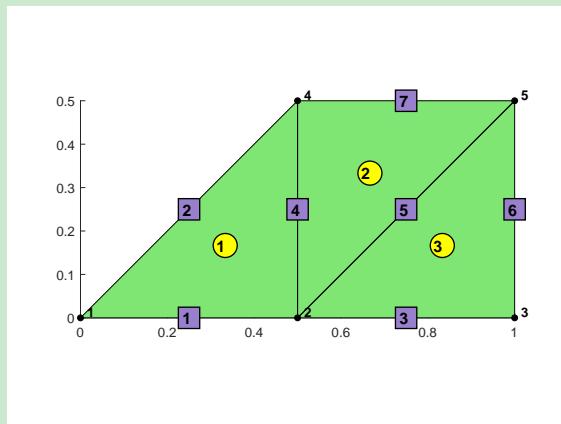
二分. 所以, 在对扩展后的标记三角形进行二分后, 再确认每个子三角形 peak 的对边不在 base 集合中, 就可给出最终的加密网格.

注 12.8 为了方便, 以下称二分标记单元 (base 的左右单元) 为标记二分, 而为了去除悬挂点的二分为协调二分.

12.4.3 标记二分的程序实现

辅助数据 `elem2edge` 按单元存储边的自然序号, 单元的 base 显然为 `elem2edge(:, 1)`. 为了方便使用 `elem2edge`, 下面编程中尽量与 `edge` 对应.

base 的生成



- base 最多 NE 个, 我们用数组 `isCutEdge` 记录. 给定初始标记单元 `elemMarked ... = [1; 3]`, 则初始的 base 如下获得

```

1  isCutEdge = false(NE,1);
2  base = elem2edge(elemMarked,1);
3  isCutEdge(base) = true;

```

其中, `elem2edge(elemMarked, 1)` 给定了标记单元第 1 条边的自然序号, 即标记单元的 base. 例子的 `base = [4; 5]`.

- 接着我们要补充相邻三角形的 base. 为此先找到相邻三角形. 辅助数据结构中给出了 `neighbor`, 它存储每个三角形边的相邻三角形序号, 据此可找到 base 的相邻三角形.

```
refineNeighbor = neighbor(elemMarked, 1);
```

注意这里用到 base 是第 1 个点的对边的规定.

- 相邻单元的 base 为

```
baseNeighbor = elem2edge(refineNeighbor,1);
```

例子的 baseNeighbor=7. 当然, 其中可能含有已存在的 base, 如下获得新的 base.

```
1 baseNeighbor = elem2edge(refineNeighbor,1);
2 elemNeighborMarked = refineNeighbor(~isCutEdge(baseNeighbor));
3 elemMarked = elemNeighborMarked;
4 base = elem2edge(elemMarked,1); isCutEdge(base) = true;
```

这里, `isCutEdge(baseNeighbor)` 确定相邻单元的 base 是否是旧的 base, 对应真的位置. 这样, `~isCutEdge(baseNeighbor)` 真的位置对应新产生的 base. 后面就是获得新产生的 base 所在的单元, 然后重复之前的标记就可获得新的 base.

- 对新单元重复之前的过程即可消除新单元可能存在的悬挂点 (可以证明, 这个过程在有限步内停止).

综上, base 如下生成

```
1 isCutEdge = false(NE,1);
2 while sum(elemMarked)>0
3     base = elem2edge(elemMarked,1); isCutEdge(base) = true;
4     refineNeighbor = neighbor(elemMarked,1);
5     baseNeighbor = elem2edge(refineNeighbor,1);
6     elemMarked = refineNeighbor(~isCutEdge(baseNeighbor));
7 end
```

`isCutEdge` 就记录了最终的 base. 例子的 `isCutEdge([4,5,7])=true`.

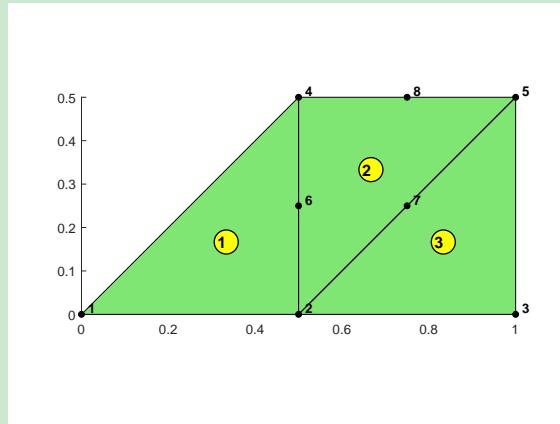
注 12.9 要想实现标记二分, 我们需要确定 base 对应的单元. 回忆一下, `elem2edge(t,1)` 给出的是单元 t 的 base 的自然序号. 于是, 可用如下语句找到单元

```
t = find(isCutEdge(elem2edge(:,1))==true);
```

标记二分的基本数据结构

现在获得标记二分的基本数据结构 node 和 elem.

对 node, 只要把标记为 base 的边的中点加入 node 即可.



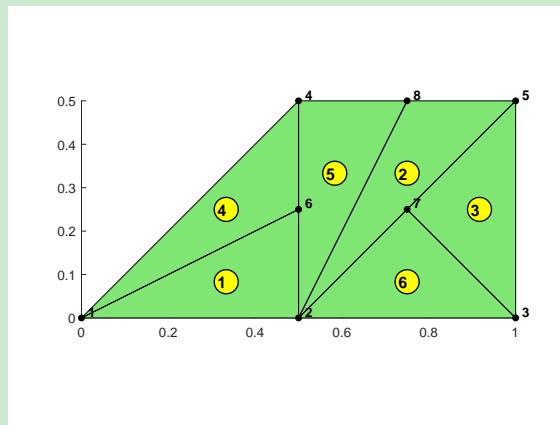
原先顶点个数为 N, base 的个数为 Nbase=`sum(isCutEdge)`. 于是可如下增加节点

```

1 % ----- Add new nodes -----
2 Nbase = sum(isCutEdge);
3 edgeCutNumber = N + (1:Nbase);
4 edgebase = edge(isCutEdge,:);
5 node(edgeCutNumber,:) = (node(edgebase(:,1),:) + node(edgebase(:,2),:))/2;

```

接着, 我们要连接 peak 和 base, 获得如下单元分布



```

1 % ----- Refine elements -----
2 edgeCutNumber = zeros(NE,1);
3 edgeCutNumber(isCutEdge) = N + (1:Nbase);
4 t = find(isCutEdge(elem2edge(:,1))==true);
5 newNT = length(t);
6 if newNT>0
7     L = t; R = NT+(1:newNT);
8     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
9     p4 = edgeCutNumber(elem2edge(t,1));
10    elem(L,:) = [p4, p1, p2];
11    elem(R,:) = [p4, p3, p1];
12 end

```

这里把 `edgeCutNumber` 对应 `edge` 给出, 是为了方便与 `elem2edge(t, 1)` 对应.

12.4.4 协调二分的程序实现

根据前面的说明, 当子三角形 peak 的对边在 base 集合中, 则该三角形要继续二分, 以去掉悬挂点. 为此, 我们要记录子三角形 peak 的对边, 即三角形的第一条边的自然序号. 显然只要存储在 `elem2edge(:, 1)` 中即可, 于是有

```
1 elem2edge(L,1) = elem2edge(t,3);
2 elem2edge(R,1) = elem2edge(t,2);
```

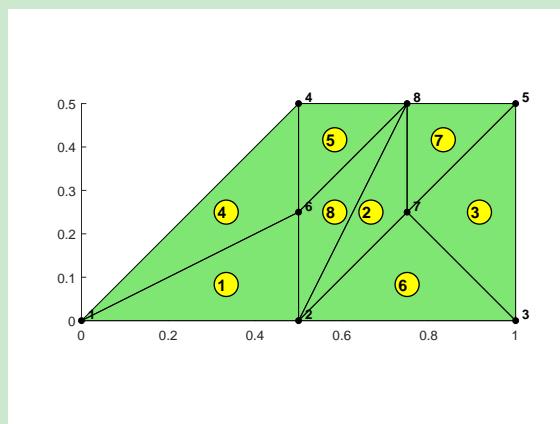
这样, 对新的剖分执行

```
t = find(isCutEdge(elem2edge(:, 1))==true);
```

就可找到需要的单元, 从而重复单元的记录即可.

```
1 % ----- Refine elements -----
2 edgeCutNumber = zeros(NE,1);
3 edgeCutNumber(isCutEdge) = N + (1:Nbase);
4 for k = 1:2
5     t = find(isCutEdge(elem2edge(:, 1))==true);
6     newNT = length(t);
7     if newNT==0, break; end
8     L = t; R = NT+(1:newNT);
9     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
10    p4 = edgeCutNumber(elem2edge(t,1));
11    elem(L,:) = [p4, p1, p2];
12    elem(R,:) = [p4, p3, p1];
13    elem2edge(L,1) = elem2edge(t,3);
14    elem2edge(R,1) = elem2edge(t,2);
15    NT = NT + newNT;
16 end
```

结果如下

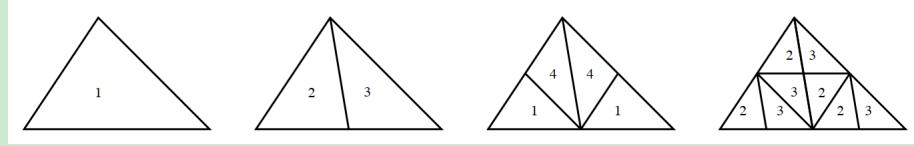


12.4.5 Newest-node bisection 程序整理

CODE 12.2. bisect.m

```
1 function [node,elem] = bisect(node,elem,elemMarked)
2 %bisect refines the elements using newest-node bisection
3 %
4 % Copyright (C) Long Chen, modified by Terence Yu.
5
6 %% Construct auxiliary data structure
7 aux = auxstructure(node,elem);
8 elem2edge = aux.elem2edge; edge = aux.edge; neighbor = aux.neighbor;
9 clear aux;
10 N = size(node,1); NT = size(elem,1); NE = size(edge,1);
11
12 %% (peak) base set
13 isCutEdge = false(NE,1);
14 while sum(elemMarked)>0
15     base = elem2edge(elemMarked,1); isCutEdge(base) = true;
16     refineNeighbor = neighbor(elemMarked,1);
17     baseNeighbor = elem2edge(refineNeighbor,1);
18     elemMarked = refineNeighbor(~isCutEdge(baseNeighbor));
19 end
20
21 %% Add new nodes
22 Nbase = sum(isCutEdge);
23 edgeCutNumber = N + (1:Nbase);
24 edgebase = edge(isCutEdge,:);
25 node(edgeCutNumber,:) = (node(edgebase(:,1),:) + node(edgebase(:,2),:))/2;
26
27 %% Refine elements
28 edgeCutNumber = zeros(NE,1);
29 edgeCutNumber(isCutEdge) = (N+1:N+Nbase)';
30 for k = 1:2
31     t = find(isCutEdge(elem2edge(:,1))==true);
32     newNT = length(t);
33     if newNT==0, break; end
34     L = t; R = NT+(1:newNT);
35     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
36     p4 = edgeCutNumber(elem2edge(t,1));
37     elem(L,:) = [p4, p1, p2];
38     elem(R,:) = [p4, p3, p1];
39     elem2edge(L,1) = elem2edge(t,3);
40     elem2edge(R,1) = elem2edge(t,2);
41     NT = NT + newNT;
42 end
```

注 12.10 Newest-node bisection 只会产生四个相似类, 如下图所示



正因为如此, 该加密满足 shape regularity.

注 12.11 bisection.m 也可用来对网格进行全局加密, 只要把所有单元视为标记单元即可.

12.5 板弯问题 Morley 元的自适应有限元方法

12.5.1 后验误差估计

定义离散范数

$$\|v\|_h^2 = \sum_{K \in \mathcal{T}_h} |v|_{2,K}^2 + \sum_{e \in \mathcal{E}_h} h_e^{-3} \| [v] \|_{0,e}^2 + \sum_{e \in \mathcal{E}_h} h_e^{-1} \| [\nabla v \cdot \vec{n}_e] \|_{0,e}^2,$$

文献 [4] 给出了如下的后验误差估计

$$\|w - w_h\|_h \lesssim \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 + \sum_{K \in \mathcal{T}_h} h_K^4 \|f - f_h\|_{0,K}^2 \right)^{1/2},$$

其中, 误差指示子

$$\begin{aligned} \eta &= \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{1/2}, \\ \eta_K^2 &= h_K^4 \|f\|_{0,K}^2 + \sum_{e \subset \partial K} c_e h_e^{-3} \| [w_h] \|_{0,e}^2 + \sum_{e \subset \partial K} c_e h_e^{-1} \| [\nabla w_h \cdot \vec{n}_e] \|_{0,e}^2, \end{aligned}$$

并且 c_e 对内部边为 $1/2$, 对边界边为 1 (为了方便,). f_h 是 f 的某种近似, 当 $f_h = f$ 时, 第二项消失.

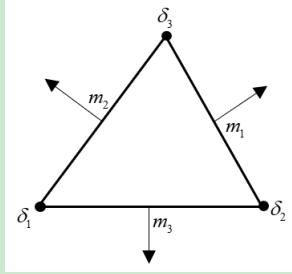
η_K^2 第一项的计算比较容易, 这里省略. 后两项的计算较为麻烦, 正因为如此, 我们在本章一开始给出了跳量计算的程序设计.

12.5.2 边上函数跳量的计算

现在我们来计算局部误差指示子的第二项

$$\sum_{e \subset \partial K} c_e h_e^{-3} \| [w_h] \|_{0,e}^2.$$

它的计算也可按照边集合进行, 但我们采用跳量计算的程序设计思路.



Morley 元的自由度含有方向, `sgnElem` 要恢复边界边的定向, 以保证插值的正确性. 这里不再详细说明计算过程, 核心部分如下

```

1 % interpolant of wh at boundary quadrature points
2 elemwhP = zeros(NT,3*ng);
3 for p = 1:3*ng
4     % basis functions at the p-th quadrature point
5     base = zeros(NT,Ndof);
6     % i = 1,2,3
7     base(:,it) = repmat(lambdaRef(p,it).^2,NT,1) ...
8             + c3.*repmat(lambdaRef(p,jt).*lambdaRef(p,kt),NT,1) ...
9             + c2.*repmat(lambdaRef(p,kt).*lambdaRef(p,it),NT,1) ...
10            + c1.*repmat(lambdaRef(p,it).*lambdaRef(p,jt),NT,1);
11    % i = 4,5,6
12    ci = 2*repmat(area,1,3)./L(:,it);
13    base(:,3+it) = ci.*repmat(lambdaRef(p,it).*(lambdaRef(p,it)-1),NT,1);
14    % equipped with global sign
15    base = sgnbase.*base;
16    % interpolant at the p-th quadrature point
17    for i = 1:Ndof
18        elemwhP(:,p) = elemwhP(:,p) + wh(elem2dof(:,i)).*base(:,i);
19    end
20 end
21 whI = zeros(2*NE*ng,1);    whI(elem2dofP) = elemwhP;
22 elemwhM = whI(elem2dofM);
23 % elementwise jump of wh
24 elem2Jumpval = (elemwhP - elemwhM).^2;
25 ce = 0.5*ones(NE,1); ce(bdEdgeIdx) = 1;
26 elemJumpval = zeros(NT,1);
27 for i = 1:3
28     hei = he(elem2edge(:,i));
29     cei = ce(elem2edge(:,i)).*hei.^(-3);
30     Jump = elem2Jumpval(:,(1:ng)+(i-1)*ng);
31     elemJumpval = elemJumpval + cei.*hei.*(Jump*weight1d(:));
32 end

```

12.5.3 边上法向导数跳量的计算

现在计算第三项

$$\sum_{e \subset \partial K} c_e h_e^{-1} \|[\nabla w_h \cdot \vec{n}_e]\|_{0,e}^2.$$

它的计算与 Poisson 方程那里类似, 只要修改基函数的导数. 核心程序如下

```

1 %% elementwise jump of whx, why --- the third term
2 % interpolant of whx, why at boundary quadrature points
3 elemwhxP = zeros(NT,3*ng);
4 elemwhyP = zeros(NT,3*ng);
5 for p = 1:3*ng
6     % derivatives of basis functions at the p-th quadrature point
7     basex = zeros(NT,Ndof); basey = zeros(NT,Ndof);
8     % i = 1,2,3
9     lambdaRefpit = repmat(lambdaRef(p,it),NT,1);
10    lambdaRefpj = repmat(lambdaRef(p,jt),NT,1);
11    lambdaRefpkt = repmat(lambdaRef(p,kt),NT,1);
12    basex(:,it) = 2*lambdaRefpit.*Dlambdax(:,it) ...
13        + c3.*(Dlambdax(:,jt).*lambdaRefpkt+lambdaRefpj.*Dlambdax(:,kt)) ...
14        + c2.*(Dlambdax(:,kt).*lambdaRefpit+lambdaRefpkt.*Dlambdax(:,it)) ...
15        + c1.*(Dlambdax(:,it).*lambdaRefpj+lambdaRefpit.*Dlambdax(:,jt));
16    basey(:,it) = 2*lambdaRefpit.*Dlambday(:,it) ...
17        + c3.*(Dlambday(:,jt).*lambdaRefpkt+lambdaRefpj.*Dlambday(:,kt)) ...
18        + c2.*(Dlambday(:,kt).*lambdaRefpit+lambdaRefpkt.*Dlambday(:,it)) ...
19        + c1.*(Dlambday(:,it).*lambdaRefpj+lambdaRefpit.*Dlambday(:,jt));
20    % i = 4,5,6
21    ci = 2*repmat(area,1,3)./L(:,it);
22    basex(:,3+it) = ci.* (Dlambdax(:,it).*(lambdaRefpit-1)+lambdaRefpit.*Dlambdax(:,it));
23    basey(:,3+it) = ci.* (Dlambday(:,it).*(lambdaRefpit-1)+lambdaRefpit.*Dlambday(:,it));
24    % equipped with global sign
25    basex = sgnbase.*basex; basey = sgnbase.*basey;
26    % interpolant at the p-th quadrature point
27    for i = 1:Ndof
28        elemwhxP(:,p) = elemwhxP(:,p) + wh(elem2dof(:,i)).*basex(:,i);
29        elemwhyP(:,p) = elemwhyP(:,p) + wh(elem2dof(:,i)).*basey(:,i);
30    end
31 end
32 whxI = zeros(2*NE*ng,1);      whyI = zeros(2*NE*ng,1);
33 whxI(elem2dofP) = elemwhxP;  whyI(elem2dofP) = elemwhyP;
34 elemwhxM = whxI(elem2dofM);  elemwhyM = whyI(elem2dofM);
35 % elementwise jump of whx, why
36 elem2Jumpx = elemwhxP - elemwhxM;
37 elem2Jumpy = elemwhyP - elemwhyM;
38 elemJumpgrad = zeros(NT,1);
39 for i = 1:3
40     hei = he(elem2edge(:,i));
41     cei = ce(elem2edge(:,i)).*hei.^(-1);
42     neix = ne(elem2edge(:,i),1); neiy = ne(elem2edge(:,i),2);

```

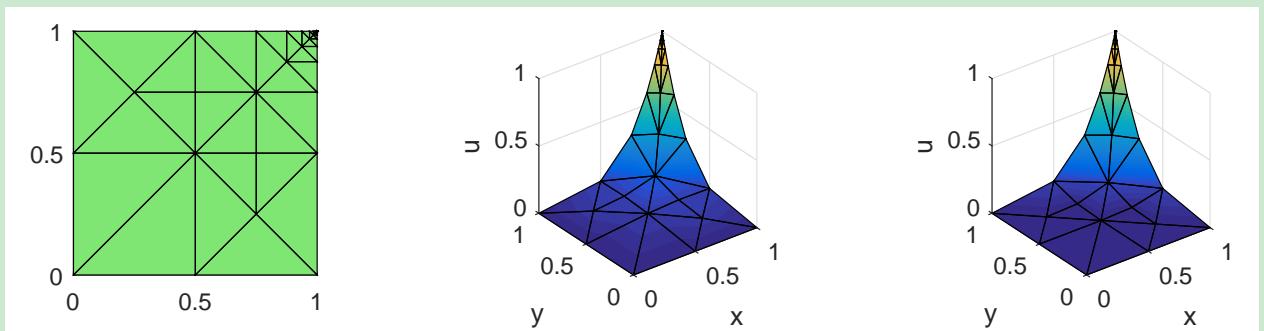
```

43     Jumpnx = elem2Jumpx(:,(1:ng)+(i-1)*ng).*repmat(neix,1,ng);
44     Jumpny = elem2Jumpny(:,(1:ng)+(i-1)*ng).*repmat(neiy,1,ng);
45     Jumpn = (Jumpnx+Jumpny).^2;
46     elemJumpgrad = elemJumpgrad + cei.*hei.* (Jumpn*weight1d(:));
47 end

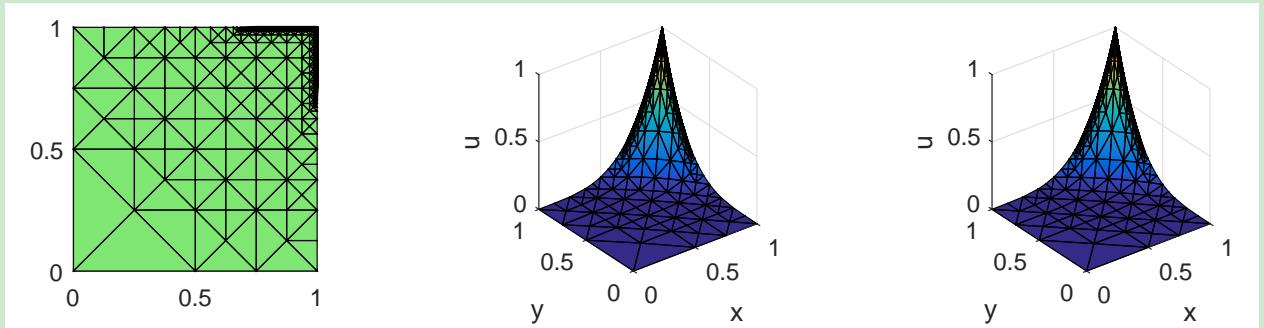
```

注 12.12 查看 Morley 元的基函数可知, 若单元过小, 则会出现非常大的因子. 这样, 局部加密不能过剩, 否则刚度矩阵容易出现局部元素很大, 从而产生奇异性. 为此, 在迭代的过程中, 只要 `edge` 的最小边长小于 `1e-8`, 我们就停止迭代.

取标记算法中的 $\theta = 0.4$, 结果如下



可以看到, 对文献 [4] 中给出的误差指示子, 加密的单元不是很多, 仅出现在函数值变化比较明显的地方. 取 $\theta = 0.99$ 的结果如下



第十三章 均匀加密网格上的多重网格法

对大规模线性代数方程组, 直接法是很难求解的. 它可认为是求矩阵的逆或其作用, 涉及到较多的稠密矩阵操作, 很容易超出计算机的内存. 为此, 我们常采用迭代法求解大规模问题. 此时仅涉及到矩阵和向量之间的基本运算, 这些操作对稀疏矩阵来说是可以接受的. 迭代法的一般理论告诉我们, 迭代的收敛速度与矩阵的谱半径或条件数相关. 对有限元方法来说, 线性方程组矩阵的条件数随网格加密而迅速增大, 经典迭代法在求解此类问题时收敛速度很慢.

本章考虑迭代法中的多重网格法 (the multigrid methods, MG methods), 它是求解大规模问题的常用算法. 为了方便, 本章仅考虑均匀加密的网格. 例如, 对三角形单元, 均匀加密指的是连接三边中点形成的剖分 (且所有三角形都进行这种加密).

13.1 嵌套有限元

13.1.1 嵌套有限元空间与子空间方程

设 Ω 是多角形区域. \mathcal{T}_1 是区域 Ω 的一个三角剖分, 网格参数为 h_1 . 将剖分进行二分, 如将三角形各边的中点进行连接得加细剖分 \mathcal{T}_2 , 相应的网格参数为 $h_2 = h_1/2$. 依此类推, 可获得一组嵌套的剖分

$$\mathcal{T}_1, \mathcal{T}_2 \cdots, \mathcal{T}_L.$$

对每个三角剖分 \mathcal{T}_l , 定义相应的连续分片线性有限元空间为

$$V_l = \left\{ v_l \in C(\overline{\Omega}) : v_l|_K \in \mathbb{P}_1(K), v_l|_{\partial\Omega} = 0 \right\}.$$

显然

$$V_1 \subset V_2 \subset \cdots \subset V_L =: V_h, \quad (13.1)$$

注意, 这里的 V_l 实际上是有限元空间 V_h 的粗化.

在 l 层上, 有限元方法为: 找 $u_l \in V_l$ 使得

$$a(u_l, v_l) = (f, v_l), \quad v_l \in V_l. \quad (13.2)$$

在空间 V_l 定义算子

$$A_l : V_l \rightarrow V_l, \quad v_l \mapsto A_l v_l,$$

满足

$$(A_l v_l, w_l) = a(v_l, w_l), \quad v_l, w_l \in V_l.$$

定义 f 在 V_l 上的 L^2 投影 $f_l \in V_l$ 为

$$(f_l, v_l) = (f, v_l), \quad v_l \in V_l,$$

则 (13.2) 可写为

$$A_l u_l = f_l.$$

设 V_l 的节点基为 $N_l = (\varphi_1, \dots, \varphi_{n_l})$, 算子 A_l 在该组基下的矩阵为 \mathbf{A}_l , 即

$$A_l N_l = A_l(\varphi_1, \dots, \varphi_{n_l}) = (\varphi_1, \dots, \varphi_{n_l}) \mathbf{A}_l = N_l \mathbf{A}_l.$$

考虑展开

$$u_l = N_l \mathbf{u}_l, \quad f_l = N_l \mathbf{f}_l,$$

则有

$$\mathbf{A}_l \mathbf{u}_l = \mathbf{f}_l.$$

显然, 上式就是有限元方程组. 在不至误会时, 我们仍用不加粗的符号表示.

注 13.1 在最细的网格空间 V_h 上, 记 $A = A_L$, 即

$$(Av_h, w_h) = a(v_h, w_h), \quad v_h, w_h \in V_h.$$

显然有

$$(A_l v_l, w_l) = a(v_l, w_l) = (Av_l, w_l), \quad v_l, w_l \in V_l. \quad (13.3)$$

注 13.2 嵌套空间 (13.1) 的 V_i 不一定是同一种有限元空间, 例如, V_h 是二次 Lagrange 元空间, 而其他层是一次 Lagrange 元空间. 之所以要强调这一点, 是因为高阶元的多重网格法经常是这样考虑的. 后面将会看到, 多重网格法实际上是在粗网格上求解残差方程, 用以校正最细的层. 正因为如此, 残差方程的精度不必和最细层一致. 当然这也是为了减少计算损耗.

13.1.2 延长、限制算子与延长、限制矩阵

设

$$V_1 \subset V_2 \subset \cdots \subset V_L$$

为嵌套空间. 定义自然嵌入

$$I_i : V_i \rightarrow V_{i+1}, \quad v_i \mapsto I_i v_i = v_i,$$

有时候也记为 $I_i = I_i^{i+1}$, 称为延长算子. 定义 L^2 投影

$$Q_i : V_{i+1} \rightarrow V_i, \quad v_{i+1} \mapsto Q_i v_{i+1},$$

满足

$$(Q_i v_{i+1}, w_i) = (v_{i+1}, w_i), \quad v_{i+1} \in V_{i+1}, \quad w_i \in V_i,$$

称其为限制算子, 也记为 $Q_i = I_{i+1}^i$. 后者记号即为转移算子.

可以证明, $Q_i = I_i^\top$, 其中 I_i^\top 表示在内积 (\cdot, \cdot) 下的伴随. 事实上,

$$(Q_i^\top v_i, v) = (v_i, Q_i v) = (v_i, v) = (I_i v_i, v) \quad \forall v_i \in V_i, v \in V_{i+1}.$$

定义 13.1 设 $N_i = (\varphi_1, \dots, \varphi_{n_i})$ 是 V_i 的节点基向量, 由 $I_i \varphi_j \in V_{i+1}$ 知, 存在矩阵 \mathbf{I}_i^{i+1} 使得

$$I_i N_i = N_{i+1} \mathbf{I}_i^{i+1}. \quad (13.4)$$

同理, 由 $Q_i \varphi_j \in V_i$, 存在矩阵 \mathbf{I}_{i+1}^i 使得

$$Q_i N_{i+1} = N_i \mathbf{I}_{i+1}^i. \quad (13.5)$$

称 \mathbf{I}_i^{i+1} 为延长矩阵, \mathbf{I}_{i+1}^i 为限制矩阵.

延长矩阵和限制矩阵用以沟通转移前后的节点值.

先考虑延长矩阵. 给定 $v \in V_i$, 它在 V_i 中的节点值向量记为 \mathbf{v}_i , $I_i v = v$ 在 V_{i+1} 中的节点值向量记为 $\hat{\mathbf{v}}_{i+1}$. 注意 $\mathcal{T}_i \subset \mathcal{T}_{i+1}$, 有 \mathbf{v}_i 是 $\hat{\mathbf{v}}_{i+1}$ 的分量. 而 $\hat{\mathbf{v}}_{i+1}$ 其余的分量是三角形边中点对应的函数值, 由线性插值知, 它们可由三角形顶点值平均得到. 于是, 存在矩阵 \mathbf{I}_i^{i+1} 使得

$$\hat{\mathbf{v}}_{i+1} = \mathbf{I}_i^{i+1} \mathbf{v}_i,$$

这里的矩阵就是前面定义的延长矩阵. 事实上, (13.4) 两边作用于 \mathbf{v}_i 有

$$I_i N_i \mathbf{v}_i = N_{i+1} \mathbf{I}_i^{i+1} \mathbf{v}_i \quad \Rightarrow \quad I_i v = N_{i+1} \mathbf{I}_i^{i+1} \mathbf{v}_i,$$

而 $I_i v = N_{i+1} \hat{\mathbf{v}}_{i+1}$, 即证.

再考虑限制矩阵. 给定 $r \in V_{i+1}$, 它在 V_{i+1} 中的节点值向量记为 \mathbf{r}_{i+1} . 因 $Q_i r \in V_i$, 设它在 V_i 中的节点值向量记为 $\tilde{\mathbf{r}}_i$, 则有

$$\tilde{\mathbf{r}}_i = \mathbf{I}_{i+1}^i \mathbf{r}_{i+1},$$

这只需要把 (13.5) 两边作用于 \mathbf{r}_{i+1} .

可以证明,

$$\mathbf{I}_{i+1}^i = (\mathbf{I}_i^{i+1})^\top.$$

13.1.3 Galerkin 条件

对两层情形, 即 $V_1 \subset V_2$, 视 V_2 为 V , 由 (13.3),

$$(A_1 u_1, v_1) = (A u_1, v_1) = (A I_1 u_1, I_1 v_1) = (I_1^\top A I_1 u_1, v_1), \quad (13.6)$$

这表明

$$A_1 = I_1^\top A I_1 = Q_1 A I_1,$$

即

$$A_1 = I_1^\top A_2 I_1 = Q_1 A_2 I_1,$$

其中, $Q_1 = I_2^1$, $I_1 = I_1^2$. 相应的矩阵形式为

$$\mathbf{A}_1 = \mathbf{I}_2^1 \mathbf{A}_2 \mathbf{I}_1^2.$$

类似有

$$\mathbf{A}_i = \mathbf{I}_{i+1}^i \mathbf{A}_{i+1} \mathbf{I}_i^{i+1}. \quad (13.7)$$

推导 (13.6) 用到双线性形式, 因而称 (13.7) 为 Galerkin 条件, 而获得 \mathbf{A}_i 的方法称为 Galerkin 方法或变分方法.

13.2 MG 的基本思想

迭代法通常有两种构造思路, 一是矩阵分裂, 一是残差或误差校正. MG 基于后者.

13.2.1 残差校正与频率分量

考虑线性算子方程

$$A u = f \quad (13.8)$$

的求解. 设 v 是 u 的一个近似, 则误差为 $e = u - v$, 残差为 $r = f - Av$. 显然误差与残差满足

$$Ae = r, \quad (13.9)$$

这样对近似解通过求解方程 (13.9) 可获得校正 $u = v + e$. 为此给出从 u_k 获得 u_{k+1} 的残差校正格式

残差校正算法

- (1) 形成残差: $r = f - Au_k$;
 - (2) 近似求解残差方程 $Ae = r$ 得校正: $\hat{e} = Br$, 其中, B 是 A^{-1} 的近似;
 - (3) 更新: $u_{k+1} = u_k + \hat{e}$.
-

算子 B 称为迭代子 (iterator) 或光滑子 (smoother). 基于残差校正的方法就是给出 A^{-1} 的一个合理近似 B , 从而可以迭代求解

$$u_{k+1} = u_k + \hat{e} = u_k + Br = u_k + B(f - Au_k).$$

对多重网格法, 我们记 $MG := B$, 从而

$$u_{k+1} = u_k + MG(f - Au_k).$$

伪代码如下

```
1 % solve Au = b
2 while Err>tolf && iter≤MaxIt
3     r = b-A*u;
4     e = MG(A,r,...);
5     u = u+e;
6     Err = norm(e); iter = iter + 1;
7 end
```

考虑问题

$$\begin{cases} -u''(x) = f(x), & x \in (0, 1), \\ u(0) = u_0, & u(1) = u_1. \end{cases}$$

使用步长为 $h = \frac{1}{N+1}$ 的均匀剖分, 易知线性元的刚度矩阵 (`kk(freeNode, freeNode)`) 为

$$A = \frac{1}{h} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & \ddots & & \\ & \ddots & \ddots & -1 & \\ & & -1 & 2 & \end{bmatrix}_{N \times N},$$

其特征值为

$$\lambda_k = \frac{4}{h} \sin^2 \frac{\theta_k}{2}, \quad k = 1, 2, \dots, N,$$

相应特征向量为

$$\mathbf{p}_k = [\sin(\theta_k), \sin(2\theta_k), \dots, \sin(N\theta_k)]^T, \quad 1 \leq k \leq N,$$

式中 $\theta_k = \frac{k}{N+1}\pi$, $1 \leq k \leq N$. 特征向量满足如下的正交关系

$$\mathbf{p}_i^T \mathbf{p}_j = \frac{1}{2h} \delta_{ij},$$

且任何一个 N 维向量都可由它展开, 自然误差和残差也是如此.

定义 13.2 称特征向量

$$\mathbf{p}_k = [\sin(\theta_k), \sin(2\theta_k), \dots, \sin(N\theta_k)]^T$$

为低频分量, 如果 $1 \leq k < \lfloor \frac{N}{2} \rfloor$. 剩下的则称为高频分量.

对小的 k , 正弦函数的波数少, 函数图像更加光滑, 所以低频分量也称为光滑分量. 高频分量函数图像摆动较大, 也称为摆动分量.

13.2.2 经典迭代法对频率分量的影响

为了方便, 考虑方程组 $Au = f$, 其中

$$A = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}.$$

设 A 分裂为 $A = D - L - U$, 其中 D 是对角线元素给出的对角矩阵, $-L$ 是下三角部分, $-U$ 是上三角部分. Jacobi 迭代可写为

$$Du^{(n+1)} = (L + U)u^{(n)} + f$$

或

$$u^{(n+1)} = D^{-1}(L + U)u^{(n)} + D^{-1}f =: Bu^{(n)} + D^{-1}f,$$

式中,

$$B = \frac{1}{2} \begin{bmatrix} 0 & 1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 0 \end{bmatrix}.$$

显然精确解 u 满足上面的迭代方程, 定义误差 $e^{(n)} = u^{(n)} - u$, 则有

$$e^{(n+1)} = Be^{(n)} = B^{n+1}e^{(0)}.$$

矩阵 B 的特征向量与 A 相同, 而特征值为

$$\lambda_k = 1 - 2\sin^2 \frac{\theta_k}{2}, \quad \theta_k = \frac{k\pi}{N+1}.$$

设初始误差展开为

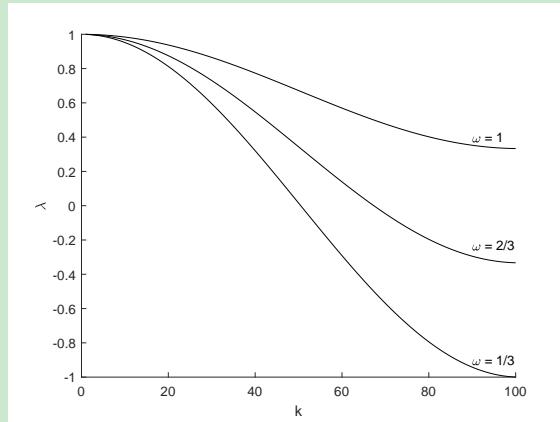
$$e^{(0)} = a_1 \mathbf{p}_1 + a_2 \mathbf{p}_2 + \cdots + a_N \mathbf{p}_N,$$

则有

$$e^{(n)} = B^n e^{(0)} = \sum_{i=1}^n a_i \lambda_i^n \mathbf{p}_i.$$

可以看到, 特征值是误差分量的放大系数. 对这里的问题, 显然有 $|\lambda_k| \leq 1$, 从而迭代收敛. 若考虑阻尼 Jacobi 迭代, 设松弛因子为 ω , 则有

$$\lambda_k(\omega) = 1 - 2\omega \sin^2 \frac{\theta_k}{2}, \quad \theta_k = \frac{k\pi}{N+1}.$$



上图给出的是不同阻尼系数特征值随波数 k 的变化. 可以看到, 不论如何选取阻尼系数 ω , 低频分量的衰减系数都不会很小. 这就是为什么经典迭代法在开始衰减较快而后面几乎停滞不动的原因. 事实上, 迭代法最开始高频分量迅速衰减, 而剩下的低频分量衰减并不明显.

注 13.3 正因为高频分量衰减得很快, 数值逼近的主要贡献是低频分量.

注 13.4 残差校正是给定 u_k 获得 u_{k+1} . 这个过程中的迭代初值 $u^{(0)} = u_k$, 残差

$$r = f - Au_k = f - Au^{(0)} = A(u - u^{(0)}) = -Ae^{(0)}.$$

消除误差 $e^{(0)} = u - u_k$ 的频率分量可等价地说消去残差 r 的频率分量.

13.2.3 两网格方法

多重网格法基于如下观察

- 误差分为高频分量和低频分量;
- 经典迭代法可消除误差的高频分量, 对低频分量的抑制作用并不明显;
- 低频部分在粗网格上就可获得较好的近似;
- 细网格上导出的方程条件数更大.

低频或高频与网格剖分有关, 细网格上的一些低频分量在粗网格上是高频分量. MG 的想法就是把细网格上的低频误差视为粗网格上的高频误差, 而对粗网格问题, 我们可再次使用光滑作用. 多重网格法的关键步骤是

- 在细网格上利用通常的迭代法, 如 Richardson 迭代, Gauss-Seidel 迭代进行迭代, 消去残差的高频部分;
- 把细网格上残差的低频部分转移到粗网格上进行残差校正 (视其为粗网格上的高频误差).

根据残差校正方法, 我们是对残差方程进行计算. 一个简单的二层网络如下

算法 3 求解残差方程的两网格算法

1. 前光滑: 在细网格 \mathcal{T}_h 上用某种迭代法求解

$$A_h e_h = r_h,$$

迭代 m_1 次获得近似解 e_h , 从而有残差方程的残差 (仍记为 r_h)

$$r_h \leftarrow r_h - A_h v_h.$$

这里 A_h 对应网格参数 h 的有限元方程.

2. 粗网格校正:

- 将细网格上的残差 r_h 转移到粗网格 \mathcal{T}_{2h} 上, 记为 $I_h^{2h} r_h$.

- 在粗网格上求解残差方程

$$A_{2h} e_{2h} = I_h^{2h} r_h,$$

获得误差 e_{2h} .

- 将粗网格上的误差 e_{2h} 转移到细网格上, 记为 $I_{2h}^h e_{2h}$, 从而获得校正

$$e_h \leftarrow e_h + I_{2h}^h e_{2h}.$$

3. 后光滑: 以 e_h 为初值, 在细网格上用某种迭代法迭代 m_2 次, 获得最终的近似解 e_h .

注 13.5 以下称转移到粗网格上为限制, 而转移到细网格上为延长, 且称开始的残差方程为原方程. 两网格方法的特点是

细网格上: 求解原方程, 即前光滑迭代, 获得残差

↓

粗网格上: 求解残差限制方程, 获得误差

↓

细网格上: 误差延长, 校正原方程的近似解, 并后光滑迭代

注 13.6 两网格方法是直接求解粗网格上的残差限制方程, 它与原方程形式相同, 只不过在粗网格上. 现在可把粗网格视为细网格, 粗网格的下一级粗网格视为粗网格, 而残差限制方程视为原方程, 从而再次使用两网格方法求解. 重复这个思想就获得多重网格法.

13.3 MG 的算法描述

13.3.1 MG 的两网格添加

根据注 13.6, 为了获得多重网格算法, 只需要逐次添加两网格.

- 1 个两网格的过程如下

$$\begin{cases} A_L e_L = r_L \Rightarrow e_L, \quad r_L \leftarrow r_L - A_L e_L \\ A_{L-1} e_{L-1} = Q_{L-1} r_L \Rightarrow e_{L-1}, \\ e_L \Leftarrow e_L + I_L e_{L-1} \end{cases},$$

最后的双箭头包含两层意思: 一是校正, 二是后光滑迭代 (仍用原符号).

- 2 个两网格的过程如下

$$\begin{cases} A_L e_L = r_L \Rightarrow e_L, \quad r_L \leftarrow r_L - A_L e_L \\ A_{L-1} e_{L-1} = Q_{L-1} r_L \left\{ \begin{array}{l} A_{L-1} e_{L-1} = r_{L-1} := Q_{L-1} r_L \Rightarrow e_{L-1}, \quad r_{L-1} \\ A_{L-2} e_{L-2} = Q_{L-2} r_{L-1} \Rightarrow e_{L-2} \\ e_{L-1} \Leftarrow e_{L-1} + I_{L-1} e_{L-2} \end{array} \right. \\ e_L \Leftarrow e_L + I_L e_{L-1} \end{cases},$$

这里视 $Q_{L-1} r_L$ 为 r_{L-1} .

由此可以看到, 在添加两网格的过程中,

- 先逐层递减求残差限制方程 $\begin{cases} Q_{L-1} \\ Q_{L-2}; \\ \vdots \end{cases}$
- 再逐层递增延长误差并后光滑 $\begin{cases} \vdots \\ I_{L-1}. \\ I_L \end{cases}$

具体来说, 多重网格法的计算过程分为如下两步:

算法 4 求解残差方程的多重网格法

Step 1: 计算每层的校正误差: 残差限制与前光滑

- L 层: 给定初值 $e_L^{(0)}$, 迭代求解 $A_L e_L = r_L$, 结果记为 e_L , (新的) 残差仍记为 r_L .
- $L-1$ 层: 以 $e_{L-1}^{(0)} = 0$ 为初值, 迭代求解 $A_{L-1} e_{L-1} = r_{L-1} := Q_{L-1} r_L$, 结果记为 e_{L-1} , 残差为 r_{L-1} .
- $L-2$ 层: 以 $e_{L-2}^{(0)} = 0$ 为初值, 迭代求解 $A_{L-2} e_{L-2} = r_{L-2} := Q_{L-2} r_{L-1}$, 所得结果记为 e_{L-2} , 残差为 r_{L-2} .
- ⋮

Step 2: 延长校正误差与后光滑

⋮

- 对 e_{L-2} 校正: $e_{L-2} \leftarrow e_{L-2} + I_{L-2} e_{L-3}$, 并以校正值为初值, 对 $A_{L-2} e_{L-2} = r_{L-2}$ 做后光滑, 结果仍记为 e_{L-2} .
 - 对 e_{L-1} 校正: $e_{L-1} \leftarrow e_{L-1} + I_{L-1} e_{L-2}$, 并以校正值为初值, 对 $A_{L-1} e_{L-1} = r_{L-1}$ 做后光滑, 结果仍记为 e_{L-1} .
 - 对 e_L 校正: $e_L \leftarrow e_L + I_L e_{L-1}$, 并以校正值为初值, 对 $A_L e_L = r_L$ 做后光滑, 结果记为 e_L , 它就是残差方程的多重网格解.
-

上面的过程可用下图表示

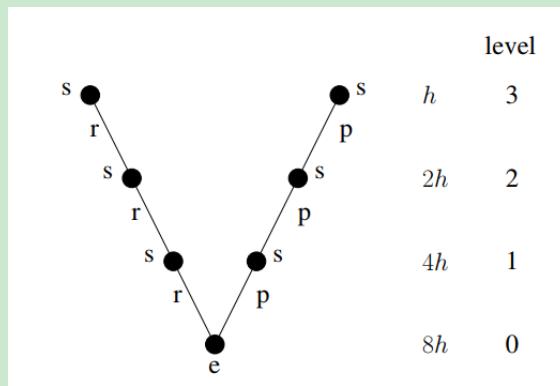


图 13.1. V 循环: s - smoothing, r - restriction, p - prolongation, e - exact solver.

这里规定: 最粗的层直接求解. 因图像形似 V, 称上面的多重网格法为 V 循环多重网格法.

13.3.2 V-循环的伪代码

根据上面的算法, V-循环的伪代码可写为

```

1 function e = Vcycle(r,J)
2 % Solve the residual equation Ae = r by multigrid V-cycle method
3
4 ri = cell(J,1);      % residual in each level
5 ei = cell(J,1);      % correction in each level
6 ri{J} = r;            % initial residual
7
8 % ----- Correction in each level -----
9 for j = J:-1:2
10    % pre-smoothing: one step
11    ei{j} = R{j}*ri{j};
12    % update and restrict residual
13    ri{j-1} = Res{j-1}*(ri{j} - Ai{j}*ei{j});
14 end
15 ei{1} = Ai{1}\ri{1}; % exact solver in the coarsest level
16
17 % ----- prolongation and correction -----
18 for j = 2:J
19    % prolongation and correction
20    ei{j} = ei{j}+Pro{j-1}*ei{j-1};
21    % post-smoothing: one step
22    ei{j} = ei{j} + R{j}'*(ri{j}-Ai{j}*ei{j});
23 end
24 e = ei{J};

```

注 13.7 后光滑中采用 R'_j 即前光滑算子 R_j 的转置, 以保证迭代矩阵 B 是对称的. 这样, B 可在预处理共轭梯度 (Preconditioned Conjugate Gradient, PCG) 法中充当预处理子. 正因为如此, 多重网格法本质上是一种预处理方法. 若在前光滑中采用 Gauss-Seidel 迭代, 即 $\mathbf{R}_j = (\mathbf{D}_j + \mathbf{L}_j)^{-1}$, 则后处理中算子转置对应的矩阵为 $(\mathbf{D}_j + \mathbf{U}_j)^{-1}$, 并不是矩阵直接转置. 前者称为 forward Gauss-Seidel, 而后者称为 backward Gauss-Seidel (这两者的区别就是前者把上三角部分视为旧变量, 后者把下三角部分视为旧变量). 注意, 这里 $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$.

注 13.8 对 2 个两网格, 在粗网格上, 我们由 e_{L-1}, r_{L-1} 获得新的 e_{L-1} . 这个过程还可以再一次执行. 事实上, 由新的 e_{L-1} 可获得 $r_{L-1} \leftarrow r_{L-1} - A_{L-1}e_{L-1}$, 从而导致需要的 e_{L-1}, r_{L-1} . 这个过程如下

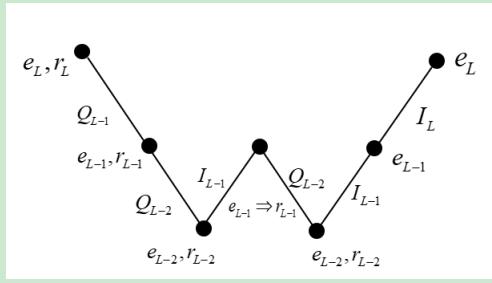


图 13.2. W 循环

该循环形似 W, 称为 W 循环多重网格法. 本文只考虑 V 循环.

13.4 多重网格法的转移矩阵

本节考虑一维问题的线性元逼近, 即嵌套有限元空间

$$V_1 \subset V_2 \cdots \subset V_L = V_h$$

中的每个空间都是一维问题的 P1-元. 后面将会看到, 高维线性元情形完全归结为一维问题的讨论.

13.4.1 延长矩阵

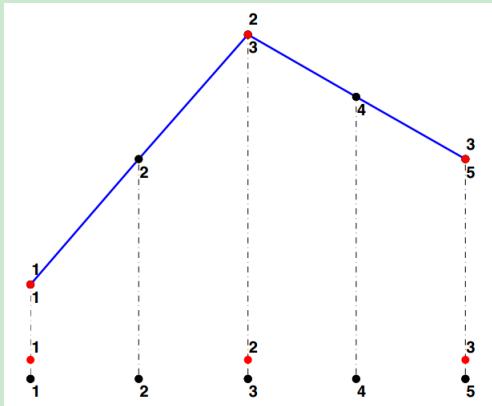


图 13.3. 延长矩阵示意图

如图, 设 3 个红色点对应的粗网格空间为 V_1 , 它是由这些节点值构造的线性有限元空间. 类似地, V_2 是由 5 个黑色点给出的有限元空间. 对 $u_1 \in V_1$, 设节点值向量为 \mathbf{u}_1 . 根据定义, $I_1^2 u_1$ 是 u_1 自然嵌入到 V_2 所得, 它们表示同一个函数, 都是 u_1 . 由 V_1 是线性有限元空间知, 在 V_2 对应的加细节点处, 函数值由 V_1 中的插值获得, 即仍在蓝色直线上. 于是

- 向量 $\mathbf{u}_1 \in \mathbb{R}^3$, $\mathbf{u}_2 = I_1^2 \mathbf{u}_1 \in \mathbb{R}^5$ 对应 V_2 中的同一个函数, 称 \mathbf{u}_2 为延长向量.

对细网格和粗网格的公共节点, 有

$$\mathbf{u}_2(1) = \mathbf{u}_1(1), \quad \mathbf{u}_2(3) = \mathbf{u}_1(2), \quad \mathbf{u}_2(5) = \mathbf{u}_1(3).$$

对细网格上的其他点, 显然延长向量满足

$$\mathbf{u}_2(2) = \frac{\mathbf{u}_1(1) + \mathbf{u}_1(2)}{2}, \quad \mathbf{u}_2(4) = \frac{\mathbf{u}_1(2) + \mathbf{u}_1(3)}{2}. \quad (13.10)$$

基于以上的式子, 我们有

$$\begin{bmatrix} \mathbf{u}_2(1) \\ \mathbf{u}_2(2) \\ \mathbf{u}_2(3) \\ \mathbf{u}_2(4) \\ \mathbf{u}_2(5) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1(1) \\ \mathbf{u}_1(2) \\ \mathbf{u}_1(3) \end{bmatrix} \Leftrightarrow \mathbf{u}_2[1, 2, 3, 4, 5] = \mathbf{I}_1^2 \mathbf{u}_1[1, 2, 3], \quad (13.11)$$

右侧方括号中是指标的顺序.

我们把细网格上的节点分为两类:

- \mathcal{C} : 粗细网格的公共节点集, 实际上就是粗网格上的节点集;
- \mathcal{F} : 只在细网格上的节点集, 即新增节点集.

为了方便, 以下在程序中规定: 公共部分, 即粗网格部分用 Coarse 标记, 细网格用 Fine 标记, 而新增的细网格部分用 f 标记.

先考虑 \mathcal{F} . 对图 13.3, \mathcal{F} 对应的编号为 $\{2, 4\}$, 节点 2 处的值可由粗网格两侧点处的值获得, 对应粗网格编号 1,2 (所在单元的左右顶点编号). 同理, 4 处对应粗网格左右的编号 2,3. 为此, 我们定义分层基 (hierarchical basis) 矩阵 HB 如下

$$\text{HB} = \begin{bmatrix} 2 & 1 & 2 \\ 4 & 2 & 3 \end{bmatrix},$$

其元素如下

- 第 1 列: 新增节点编号 (粗区间中点编号);
- 第 2 列: 新增节点所在粗单元的左顶点编号;
- 第 3 列: 新增节点所在粗单元的右顶点编号.

显然 2,3 列对应第 1 列的插值节点 (注意上面的编号都是细网格的).

对集合 \mathcal{C} , 尽管它们是公共节点, 但在粗细网格上的编号并不相同. 为此, 我们需要定义一个指标映射 Coarse2Fine , 它把粗网格上的编号映射为细网格上的. 图 13.3 中给出的

```
Coarse2Fine = [1 3 5]';
```

注意, 这里索引与粗网格节点一一对应.

可利用 `sparse` 命令生成延长矩阵 \mathbf{I}_1^2 . 显然 \mathbf{I}_1^2 的行号与细网格节点对应, 列号与粗网格节点对应.

- 对公共部分, 细网格的编号为 `Coarse2Fine`, 粗网格为 `CoarseId`, 相应的节点值相同, 从而贡献值为 1.
- 根据插值公式 (13.10), 细网格编号 $\text{HB}(:, 1)$ 对应的左侧插值节点编号为 $\text{HB}(:, 2)$, 相应的贡献值为 0.5;
- 细网格编号 $\text{HB}(:, 1)$ 对应的右侧插值节点编号为 $\text{HB}(:, 3)$, 相应的贡献值为 0.5.

需要注意的是, 对新增节点来说, 左右是相对的, 因为加法没有顺序, 所以哪边是左无所谓, 只要序号正确即可.

这样, 延长矩阵如下获得.

```
1 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
2 jj = [CoarseId; HB(:,2); HB(:,3)];
3 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
4 Pro = sparse(ii,jj,ss,nFine,nCoarse);
```

在这段代码中,

- 延长矩阵的行 `ii` 对应细网格, 列 `jj` 对应粗网格.
- `ii`, `jj`, `ss` 元素如下

	公共部分	新增部分	新增部分
ii	细网格编号	新增细节点编号	新增细节点编号
jj	粗网格编号	插值左端点编号	插值右端点编号
ss	值为 1	值为 0.5	值为 0.5

- `nCoarse` 是粗网格节点的个数, `nFine` 是细网格节点的个数, `nf` 是新增节点的个数.

有了延长矩阵 `Pro`, 则限制矩阵 `Res = Pro'`, 从而可得每层的刚度矩阵. 延长矩阵和限制矩阵统称为转移矩阵或转移算子.

13.4.2 转移矩阵的加密获取

给定初始剖分, 我们可生成 J 个嵌套剖分 (含初始), 但计算中只需要存储最后一个剖分的 node, elem.

1. 新剖分通过对上一个剖分的单元添加中点获得, 并接着顶点的序号进行编号. 如下生成最后一个剖分

```
1 a = 0; b = 1;
2 N0 = 3; x = linspace(a,b,N0)'; % N0: number of initial nodes
3 node = x; elem = [(1:N0-1)', (2:N0)']; % initial mesh
4 for j = 2:J
5
6     N = size(node,1); NT = size(elem,1); Ndof = 2;
7     % add new nodes
8     node(N+1:N+NT) = (node(elem(:,1))+node(elem(:,2)))/2;
9
10    % add new elements
11    % 1 --- 3 --- 2
12    t = 1:NT; p = zeros(NT,2*Ndof);
13    p(:,1:2) = elem; p(:,3) = (1:NT)' + N;
14    elem(t,:) = [p(t,1), p(t,3)];
15    elem(NT+1:2*NT,:) = [p(t,3), p(t,2)];
16 end
```

区域左右端点的编号就是初始剖分对应的编号, 即左端点编号为 1, 右端点编号为 N_0 .

2. 对上面的循环, 当前情形下, 分层基矩阵如下获得

```
1 % HB in the current level
2 HB = zeros(NT,3);
3 HB(:,1) = (1:NT)' + N; % 新增节点编号
4 HB(:,2:3) = elem; % 新增节点左右节点编号
```

这里, elem 是旧的单元信息.

3. 由于我们是在原有剖分节点的基础上继续编号, 故公共节点处的编号不变, 从而有

```
1 % Prolongation matrix
2 Coarse2Fine = (1:N)'; CoarseId = (1:N)';
3 nCoarse = N; nf = nel; nFine = nCoarse+nf;
4 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
5 jj = [CoarseId; HB(:,2); HB(:,3)];
6 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
7 Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
8 Res{j-1} = Pro{j-1}';
```

上面生成了最终的网格剖分、延长矩阵和限制矩阵，为了方便，编写为函数 MeshVcycle.m.

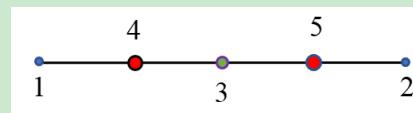
```

1 function [node,elem,Pro,Res] = Mesh1DVcycle(node,elem,J)
2
3 if J≤1
4     Pro = []; Res = []; return;
5 end
6
7 Pro = cell(J-1,1); Res = cell(J-1,1);
8 for j = 2:J
9
10    N = size(node,1); nel = size(elem,1);
11    % HB in the current level
12    HB = zeros(nel,3);
13    HB(:,1) = (1:nel)'+N; % number of the new nodes
14    HB(:,2:3) = elem;      % number of the left and right nodes
15
16    % Prolongation matrix
17    Coarse2Fine = (1:N)'; CoarseId = (1:N)';
18    nCoarse = N; nf = nel; nFine = nCoarse+nf;
19    ii = [Coarse2Fine; HB(:,1); HB(:,1)];
20    jj = [CoarseId; HB(:,2); HB(:,3)];
21    ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
22    Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
23    Res{j-1} = Pro{j-1}';
24
25    % Refine the mesh
26    [node,elem] = uniformrefine1(node,elem);
27 end

```

13.4.3 转移矩阵的粗化获取

上面是一种顺向过程，我们更常见的是给定二分加密的网格，然后推出 Pro 和 Res。给定初始网格信息 node 和 elem，设 uniformrefine1 加密 $J-1$ 次。例如，下图是 $\text{elem} = \dots [1 2]$ 加密两次所得网格。



注意，加密 $J-1$ 次，层数为 J 。

- 考虑第 J 层，称第 $J-1$ 层为粗网格。加密的过程是添加粗网格单元的中点，这些新增节点接着原先节点进行编号，顺序为粗网格单元的自然序号。另外，uniformrefine1.m 在加密的过程中，先排列所有左单元，再排列右单元。

- 由此可知 HB 为

```

1 %% HB in the level J-1
2 N = length(unique(elem(:))); nel = size(elem,1);
3 nel = nel/2; % number of coarse mesh
4 t1 = 1:nel; t2 = t1+nel;
5 HB = zeros(nel,3);
6 HB(:,1) = elem(t1, 2); % number of the new nodes
7 elem = [elem(t1, 1), elem(t2, 2)]; % coarse mesh
8 HB(:,2:3) = elem;

```

- 转移矩阵为

```

1 %% Transfer matrices in the level J-1
2 Coarse2Fine = unique(elem(:)); CoarseId = Coarse2Fine;
3 nCoarse = length(CoarseId); nFine = N; nf = size(HB,1);
4 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
5 jj = [CoarseId; HB(:,2); HB(:,3)];
6 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
7 Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
8 Res{j-1} = Pro{j-1}';

```

以上说明将总结为函数文件 uniformtransferoperator1.m, 这里的 uniform 与 uniformrefine1 对应. 注意, 该函数的输入要指定加密的层数 J :

```
1 [Pro,Res] = uniformtransferoperator1(elem,J);
```

而且该函数不必传入 node 信息.

13.5 一维问题的 MG 方法

13.5.1 刚度矩阵和载荷向量

直接法求解只考虑自由节点部分, 即

```
1 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode); % direct
```

MG 方法则要整体考虑, 因为新增的点要用到边界值进行平均. 为此, 要保留 Dirichlet 节点变量, 做法就是直接恒等替换 (并保持对称), 如下

```

1 A = speye(N); A(freeDof,freeDof) = kk(freeDof,freeDof);
2 b = u; b(freeDof) = ff(freeDof);
3 u = mgVcycle(A,b,Pro,Res); % multigrid Vcycle

```

这里的 A, b 就是 MG 求解的矩阵和右端.

完整的程序为

```

1 function u = FEM1DVcycle(node,elem,pde,bdStruct,Pro,Res)
2 %FEM1DVcycle solves the 1-D partial diffential equation by Multigrid
3 % V-cycle method
4 %
5 % -au''+bu+cu = f, x \in (x0,xL);
6 % g_N = du(x0), g_D = u(xL);
7 % or g_D = u(x0), g_N = du(xL);
8 %
9 % where, g_N for Neumann boundary conditions and
10 % g_D for Dirichlet boundary conditions,
11 % a, b and c are constants.
12 %
13 % Copyright (C) Terence Yu.
14
15 N = size(node,1); Ndof = 2;
16
17 %% Assemble stiffness matrix
18 % All element matrices
19 para = pde.para;
20 a = para.a; b = para.b; c = para.c;
21 x1 = node(elem(:,1)); x2 = node(elem(:,2));
22 h = x2-x1;
23 k11 = a./h+b/2*(-1)+c.*h./6*2;
24 k12 = a./h*(-1)+b/2+c.*h./6;
25 k21 = a./h*(-1)+b/2*(-1)+c.*h./6;
26 k22 = a./h+b/2+c.*h./6*2;
27 K = [k11,k12,k21,k22]; % stored in rows
28 % stiffness matrix
29 ii = reshape(repmat(elem, Ndof,1), [], 1);
30 jj = repmat(elem(:,1), Ndof, 1);
31 kk = sparse(ii,jj,K(:,N,N));
32
33 %% Assemble load vector
34 xc = (x1+x2)./2;
35 F1 = pde.f(xc).*h./2; F2 = F1; F = [F1,F2];
36 ff = accumarray(elem(:,1), F(:,1));
37
38 %% Assemble Neumann boundary conditions
39 Neumann = bdStruct.Neumann;
40 if ~isempty(Neumann)
41     nvec = 1;
42     if find(elem(:,1)==Neumann), nvec = -1; end
43     Dnu = pde.Du(node(Neumann,:))*nvec;
44     ff(Neumann) = ff(Neumann) + a*Dnu;
45 end
46
47 %% Apply Dirichlet boundary conditions
48 Dirichlet = bdStruct.Dirichlet; g_D = pde.g_D;
49 isBdNode = false(N,1); isBdNode(Dirichlet) = true;

```

```

50 bdDof = (isBdNode); freeDof = (~isBdNode);
51 u = zeros(N,1); u(bdDof) = g_D(node(Dirichlet));
52 ff = ff - kk*u;
53
54 %% Set solver
55 disp('Multigrid V-cycle Preconditioner with Gauss-Seidel Method');
56 fprintf('\n');
57 A = speye(N); A(freeDof,freeDof) = kk(freeDof,freeDof);
58 b = u; b(freeDof) = ff(freeDof);
59 u = mgVcycle(A,b,Pro,Res); % multigrid Vcycle

```

注 13.9 后面将给出 V-循环的程序, 要注意它是通用的, 对高维问题也是如此, 只需要获取 A 和 b 即可.

13.5.2 多重网格函数 mgVcycle.m

再次声明一下, 本小节给出的 V-循环函数适用于所有线性元问题. 实际上, 它适用于所有问题, 因为程序中只涉及到转移矩阵.

mgVcycle 函数

前面给出了 V-循环的伪代码, 用到延长矩阵、限制矩阵, 每层的系数矩阵以及光滑迭代. 有了 Pro , Res , 就可生成每层的矩阵 A_i . mgVcycle.m 函数如下

```

1 function u = mgVcycle(A,b,Pro,Res)
2
3 J = length(Pro)+1;
4 Ai = cell(J,1); Ai{J} = A; % matrices in subspaces
5 % note that J-1:-1:1 is empty when J<=1 (solve it directly)
6 for j = J-1:-1:1
7     Ai{j} = Res{j}*Ai{j+1}*Pro{j};
8 end
9
10 tol = 1e-6; tol_f = tol * norm(b); % Relative tolerance
11 Err = 10; u = zeros(size(b));
12 iter = 0; MaxIt = 20;
13 while Err>tol_f && iter<=MaxIt
14     r = b-A*u;
15     e = Vcycle(A,r,Ai,Pro,Res);
16     u = u+e;
17     Err = norm(e); iter = iter + 1;
18 end

```

注意, 这里的循环表示误差校正.

Vcycle 函数

现在伪代码可如下具体化

```
1 function e = Vcycle(A,r,Ai,Pro,Res)
2 % Solve the residual equation Ae = r by multigrid V-cycle method
3
4 J = length(Ai); % level length
5
6 % If the problem is small enough, solve it directly
7 if J≤1
8     e = A\r;      return;
9 end
10
11 ri = cell(J,1);           % residual in each level
12 ei = cell(J,1);           % correction in each level
13 ri{J} = r;
14
15 % ----- Correction in each level -----
16 option = 'forward';
17 for j = J:-1:2
18     % % pre-smoothing: one step
19     % ei{j} = R{j}*ri{j};
20     ei{j} = smoother(Ai{j},ri{j},option);
21     % update and restrict residual
22     ri{j-1} = Res{j-1}*(ri{j}-Ai{j}*ei{j});
23 end
24 ei{1} = Ai{1}\ri{1}; % exact solver in the coarsest level
25
26 % ----- prolongation and correction -----
27 option = 'backward';
28 for j = 2:J
29     % prolongation and correction
30     ei{j} = ei{j}+Pro{j-1}*ei{j-1};
31     % % post-smoothing: one step
32     % ei{j} = ei{j} + R{j}'*(ri{j}-Ai{j}*ei{j});
33     rij = ri{j}-Ai{j}*ei{j};
34     ei{j} = ei{j} + smoother(Ai{j},rij,option);
35 end
36 e = ei{J};
```

smoother 函数

前后光滑分别用 forward Gauss-Seidel 和 backward Gauss-Seidel, 程序如下

```
1 function ei = smoother(Ai,ri,option)
2 switch option
3     case 'forward'
4         Ri = tril(Ai); % Forward Gauss-Seidel    R = D+L
```

```

5      case 'backward'
6          Ri = triu(Ai); % Backward Gauss-Seidel R = D+U
7 end
8 ei = Ri\ri;

```

注意, Vcycle.m 和 smoother.m 都放置在 mgVcycle.m 中.

13.5.3 主程序

主程序如下

```

1 clc; clear; close all;
2 %% Parameters
3 maxIt = 5;
4 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
5 ErrL2 = zeros(maxIt,1);
6 ErrH1 = zeros(maxIt,1);
7
8 %% Generate an intial mesh
9 a = 0; b = 1; N0 = 3;
10 node = linspace(a,b,N0)'; % uniform
11 elem = [(1:N0-1)', (2:N0)'];
12 bdNeumann = 'abs(x-1)<1e-4';
13
14 %% Get the PDE data
15 a = 1; b = 1; c = 0;
16 para = struct('a',a, 'b',b, 'c',c);
17 pde = pde1D(para);
18
19 %% Finite element method
20 for k = 1:maxIt
21     % refine mesh
22     [node,elem] = uniformrefine1(node,elem);
23     % set boundary
24     bdStruct = setboundary1(node,elem,bdNeumann);
25     % set up solver type
26     option.solver = 'mg';
27     option.J = k+1;
28     % solve the equation
29     uh = FEM1D(node,elem,pde,bdStruct,option);
30     % record
31     NNdof(k) = length(uh);
32     h(k) = 1/size(elem,1);
33     if NNdof(k) < 1e2 % show solution for small size
34         figure(1);
35         [node1,id] = sort(node);
36         plot(node1,uh(id),'r-',[node1,pde.uexact(node1), ...
37             'k*', 'linewidth',2];
38         pause(1);
39     end

```

```

40      % compute error
41      ErrL2(k) = getL2error1(node, elem, pde.uexact, uh);
42      ErrH1(k) = getH1error1(node, elem, pde.Du, uh);
43 end
44
45 %% Plot convergence rates and display error table
46 figure(2);
47 showrateh(h, ErrH1, ErrL2);
48
49 fprintf('\n');
50 disp('Table: Error')
51 colname = {'#Dof', 'h', '|u-u_h|_1', '|u-u_h|_1'};
52 disptable(colname, NNdof, [], h, '%0.3e', ErrL2, '%0.5e', ErrH1, '%0.5e');
53
54 %% Conclusion
55 %
56 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
57 % (2nd order) is observed.

```

这里是用粗化获取转移矩阵, 程序名为 main_FEM1DVcycle_coarsening. 也可使用加密方式计算, 主程序为 main_FEM1DVcycle_refining.

13.6 二维问题的 MG 方法

仍考虑 Poisson 方程的线性元问题.

13.6.1 归结为一维问题

这一节考虑二维问题, 我们将看到, 后面的处理几乎是按照一维问题逐字逐句照抄 (对三维问题的线性元也是如此). 我们简单分析一下原因.

- HB, Pro 和 Res 涉及的是剖分顶点以及加细给出的节点. 对二维问题, 可以把所有一维边视为一维问题的一个个“单元”. 粗网格的节点就是“单元”的左右顶点, 而加细节点就是“单元”的中点. 正因为如此, 从边集合角度来看, 二维问题完全归结为一维问题 (三维问题显然也是如此).
- 刚度矩阵和载荷向量的处理也与一维问题相同, 只需要保留 Dirichlet 节点变量, 即采用恒等替换法给出 A 和 b, 如下

```

1 % -----
2 %u(freeNode) = kk(freeNode,freeNode)\ff(freeNode); % direct
3 A = speye(N); A(freeNode,freeNode) = kk(freeNode,freeNode);
4 b = u; b(freeNode) = ff(freeNode);
5 u = mgVcycle(A,b,Pro,Res); % multigrid Vcycle

```

这里的 mgVcycle 也不用变.

- 我们唯一要做的就是给出网格加密, 同时生成 `Pro` 和 `Res` (视边集合为一维问题的单元).

13.6.2 转移矩阵的加密获取

现在, 我们要给定一个初始网格, 然后连接每个三角形边的中点获得加密剖分 (可以是其他加密), 这种加密称为正规加密或一致加密.

正规加密很容易实现. 显然对一维边进行编号就可给出三角形边中点的编号. 在辅助数据结构中, 我们给出了两个与边相关的数据结构, 它们分别是

- `edge`: 一维边的端点标记;
- `elem2edge`: 边的自然序号 (按单元存储).

由此可给出加密. 在循环加密过程中, 我们可给出需要的延长矩阵和限制矩阵, 程序如下

```

1 function [node, elem, Pro, Res] = Mesh2DVcycle(node, elem, J)
2
3 if J≤1
4     Pro = []; Res = []; return;
5 end
6
7 Pro = cell(J-1,1); Res = cell(J-1,1);
8 for j = 2:J
9     % auxiliary mesh data
10    aux = auxstructure(node, elem);
11    edge = aux.edge; elem2edge = aux.elem2edge;
12    N = size(node,1); NT = size(elem,1); NE = size(edge,1); Ndof = 3;
13
14    % HB in the current level
15    HB = zeros(NE,3);
16    HB(:,1) = (1:NE)'+N; HB(:,2:3) = edge;
17
18    % Prolongation matrix
19    Coarse2Fine = (1:N)'; CoarseId = (1:N)';
20    nCoarse = N; nf = NE; nFine = nCoarse+nf;
21    ii = [Coarse2Fine; HB(:,1); HB(:,1)];
22    jj = [CoarseId; HB(:,2); HB(:,3)];
23    ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
24    Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
25    Res{j-1} = Pro{j-1}';
26
27    % add new nodes: middle points of all edges
28    node(N+1:N+NE,:) = (node(edge(:,1),:)+node(edge(:,2),:))/2;
29
30    % add new elements: refine each triangle into four triangles as follows
31    % 3

```

```

32      % | \
33      % 5- 4
34      % |\ | \
35      % 1- 6- 2
36      t = 1:NT; p = zeros(NT,6);
37      p(:,1:3) = elem;
38      p(:,4:6) = elem2edge + N;
39      elem(t,:) = [p(t,1), p(t,6), p(t,5)];
40      elem(NT+1:2*NT,:) = [p(t,6), p(t,2), p(t,4)];
41      elem(2*NT+1:3*NT,:) = [p(t,5), p(t,4), p(t,3)];
42      elem(3*NT+1:4*NT,:) = [p(t,4), p(t,5), p(t,6)];
43 end

```

上面的过程可以说是直白的, 不再解释.

13.6.3 转移矩阵的粗化获取

类似一维, 我们要根据 uniformrefine.m 的特点进行粗化以获得转移矩阵, 相应的函数命名为 uniformtransferoperator.m. 均匀加密的结构如下图

```

1 % 3                                % 3
2 % | \                                % | \
3 % 5- 4                                % |   \
4 % |\ | \                                % |   |
5 % 1- 6- 2                                % 1- - 2
6 %
7 % elem: [1 6 5],      [6 2 4],      [5 4 3],      [4 5 6]
8 %      1:NT,      NT+1:2*NT,      2*NT+1:3*NT,      3*NT+1:4*NT

```

- HB 的第一列对应中间的三角形 4-5-6 的顶点, 相应的左右端点可由其他三角形获得. 注意, 加密过程中先存储所有三角形 1-6-5, 再存所有的 6-2-4, 5-4-3 和 4-5-6. 由此可知, HB 为

```

1 %% HB in the level J-1
2 N = length(unique(elem(:))); NT = size(elem,1);
3 NT = NT/4; % number of coarse elements
4 t1 = 1:NT; t2 = t1+NT; t3 = t2+NT; t4 = t3+NT;
5 HB = zeros(3*NT,3); % elementwise HB
6 HB(:,1) = reshape(elem(t4,:),[],1);
7 elem = [elem(t1,1), elem(t2,2), elem(t3,3)]; % coarse mesh
8 HB(:, 2) = reshape(elem(:,[2,3,1]),[],1);
9 HB(:, 3) = reshape(elem(:,[3,1,2]),[],1);
10 [~,idx] = unique(HB(:,1));
11 HB = HB(idx,:);

```

注意程序中, 我们逐个单元存储所有新增节点的 HB 信息, 最后去除重复的.

- 转移矩阵与一维基本一致, 如下

```

1 %% Transfer matrices in the level J-1
2 Coarse2Fine = unique(elem(:)); CoarseId = Coarse2Fine;
3 nCoarse = length(CoarseId); nFine = N; nf = size(HB,1);
4 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
5 jj = [CoarseId; HB(:,2); HB(:,3)];
6 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
7 Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
8 Res{j-1} = Pro{j-1}';

```

13.6.4 主程序

这里仅给出粗化的主程序.

CODE 13.1. main_PoissonVcycle_coarsening.m

```

1 clc; clear; close all;
2 %% Parameters
3 maxIt = 5;
4 h = zeros(maxIt,1); NNdof = zeros(maxIt,1);
5 ErrL2 = zeros(maxIt,1);
6 ErrH1 = zeros(maxIt,1);
7
8 %% Generate an intial mesh
9 [node,elem] = squaremesh([0 1 0 1],1,1);
10 bdNeumann = 'abs(x-1)<1e-4';
11
12 %% Get the PDE data
13 pde = Poisondatal();
14
15 %% Finite element method
16 for k = 1:maxIt
17     % refine mesh
18     [node,elem] = uniformrefine(node,elem);
19     % set boundary
20     bdStruct = setboundary(node,elem,bdNeumann);
21     % set up solver type
22     option.solver = 'mg';
23     option.J = k+1;
24     % solve the equation
25     uh = Poisson(node,elem,pde,bdStruct,option);
26     % record
27     NNdof(k) = length(uh);
28     h(k) = 1/(sqrt(size(node,1))-1);
29     if NNdof(k)<2e3
30         figure(1);
31         showresult(node,elem,pde.uexact,uh);
32         pause(1);

```

```

33     end
34     % compute error
35     ErrL2(k) = getL2error(node, elem, pde.uexact, uh);
36     ErrH1(k) = getH1error(node, elem, pde.Du, uh);
37 end
38
39 %% Plot convergence rates and display error table
40 figure(2);
41 showrateh(h, ErrH1, ErrL2);
42
43 fprintf('\n');
44 disp('Table: Error')
45 colname = {'#Dof', 'h', '| |u-u_h| |', '| u-u_h |_1'};
46 disptable(colname, NNdof, [], h, '%0.3e', ErrL2, '%0.5e', ErrH1, '%0.5e');
47
48 %% Conclusion
49 %
50 % The optimal rate of convergence of the H1-norm (1st order), L2-norm
51 % (2nd order) is observed.

```

13.7 向量有限元与高阶元的 MG 方法

13.7.1 向量有限元的 MG 方法

对向量有限元, 我们采用 $A = (A_{ij})$ 形式的分块. 显然只要把 Pro 和 Res 复制成分块对角矩阵, 块的个数与变量个数相同. 以 Navier 形式为例, 最后的求解器部分变为

```

1 %% Set solver
2 disp('Multigrid V-cycle Preconditioner with Gauss-Seidel Method');
3 fprintf('\n');
4 Pro = cellfun(@(Pro) blkdiag(Pro, Pro), Pro, 'UniformOutput', false);
5 Res = cellfun(@(Res) blkdiag(Res, Res), Res, 'UniformOutput', false);
6 A = speye(2*N); A(freeDof, freeDof) = kk(freeDof, freeDof);
7 b = u; b(freeDof) = ff(freeDof);
8 u = mgVcycle(A, b, Pro, Res); % multigrid Vcycle

```

这里使用 cellfun 函数实现元胞数组每个元素的相同操作.

函数文件命名为 elasticityNavierVcycle.m, 相应的主程序这里也不再给出, 与 Poisson 方程类似.

13.7.2 二次和三次 Lagrange 元的 MG 方法

我们再回顾一下残差校正算法. 从 u_k 获得 u_{k+1} 的残差校正格式为

-
- (1) 形成残差: $r = f - Au_k$;
- (2) 近似求解残差方程 $Ae = r$ 得校正: $\hat{e} = Br$, 其中, B 是 A^{-1} 的近似;
- (3) 更新: $u_{k+1} = u_k + \hat{e}$.
-

- 可以看到, 迭代过程中的关键步骤是求解残差方程. 由于它与原方程形式一致, 高效求解就必须寻找 A^{-1} 的好的近似 B .
- 多重网格法就给出了这样的近似, 体现在如下代码中的 Vcycle 中.

```

1 function u = mgVcycle(A,b,Pro,Res)
2 ...
3
4 while Err>tolf && iter<=MaxIt
5     r = b-A*u;
6     e = Vcycle(A,r,Ai,Pro,Res);
7     u = u+e;
8     Err = norm(e); iter = iter + 1;
9 end

```

- Vcycle 考虑嵌套有限元空间

$$V_1 \subset V_2 \subset \cdots \subset V_L := V_h.$$

对二阶元, V_h 就是 P2-Lagrange 元空间. 按照正常思路, 我们似乎将每个 V_i 都选为相应剖分的 P2-Lagrange 元空间. 这种做法是不可取的, 因为限制在每个子空间上, 我们仍要求解相同的问题. 直接使用高次元的计算开销是相当大的. 为此, 通常的做法是将所有粗网格的有限元空间选为 P1-元.

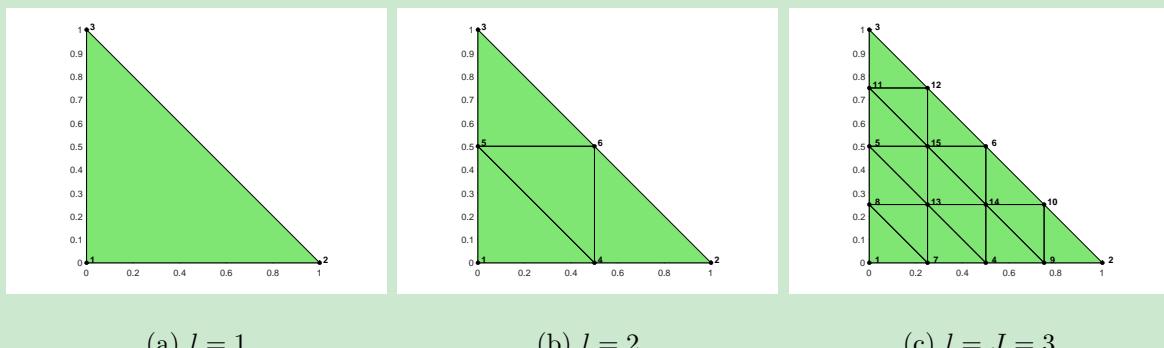


图 13.4. 三层均匀网格

考虑图 13.4 中的网格. 在第三层网格上使用 P2-Lagrange 元, 此时每个小三角形边的中点是自由度. 我们将使用边顶点的值进行平均近似, 为此必须事先算出 J 层网格的所有顶点值. 这表明, 分层考虑时, 第 J 层网格也必须纳入线性元中. 也就是说, 我们考虑的仍是如下的嵌套

$$V_1 \subset V_2 \subset \cdots \subset V_L \subset V_h, \quad L = J,$$

这里的每个 V_i 都是线性元, 但 V_h 并不是 V_L , 只不过 V_L 和 V_h 在同一层罢了. 正因为如此, `Pro` 和 `Res` 将多出一个元胞分量, 用以沟通 V_L 和 V_h , 实现 P1 与 P2 的转换. P2-元的转移矩阵的处理总结为函数 `uniformtransferoperatorP2.m`, 多出的分量如下计算.

```

1 Pro = cell(J,1); Res = cell(J,1);
2
3 %% In the level J-1
4 allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
5 totalEdge = sort(allEdge,2);
6 edge = unique(totalEdge,'rows');
7 N = max(elem(:)); NE = size(edge,1);
8 ii = [(1:N)';      % vertices
9      (1:NE)' + N; % 1/2-v1
10     (1:NE)' + N; % 1/2-v2
11 ];
12 jj = [(1:N)';      % vertices
13     edge(:)      % 1/2-v1, 1/2-v2
14 ];
15 ss = [ones(N,1); 0.5*ones(2*NE,1)];
16 Pro{J} = sparse(ii,jj,ss,N+NE,N);
17 Res{J} = Pro{J}';

```

- 这里标注的 $1/2\text{-v1}$ 表示边中点左侧的顶点, $1/2\text{-v2}$ 则是右侧顶点.
- 注意, `mgVcycle` 仍是通用的, 因为它本身只涉及到转移矩阵.
- `Vcycle` 的求解器已放置在 `PoissonP2.m` 中.

P3-Lagrange 元的局部自由度有 10 个, 排列为:

$$\begin{cases} v(z_i), & i = 1, 2, 3; \\ v(a_i), & i = 1, 2, 3; \\ v(b_i), & i = 1, 2, 3; \\ v(z_c). \end{cases}$$

这里, a_i 第 i 条边的 $1/3$ 点, b_i 则是 $2/3$ 点, 而 z_c 是单元的重心. 类似可知 P1 到 P2 的转化部分如下.

```

1 Pro = cell(J,1); Res = cell(J,1);
2
3 %% In the level J-1
4 allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
5 totalEdge = sort(allEdge,2);
6 edge = unique(totalEdge, 'rows');
7 N = max(elem(:)); NE = size(edge,1); NT = size(elem,1);
8 % HB in the level J-1
9 ii = [(1:N)';           % vertices
10    (1:NE)' + N;        % 1/3-v1
11    (1:NE)' + N;        % 1/3-v2
12    (1:NE)' + N+NE;    % 2/3-v1
13    (1:NE)' + N+NE;    % 2/3-v2
14    (1:NT)' + N+2*NE; % center-v1
15    (1:NT)' + N+2*NE; % center-v2
16    (1:NT)' + N+2*NE % center-v3
17    ];
18 jj = [ (1:N)';       %vertices
19    edge(:); % 1/3-v1, 1/3-v2
20    edge(:); % 2/3-v1, 2/3-v2
21    elem(:) % center-v1,v2,v3
22    ];
23 ss = [ones(N,1); % vertices
24    2/3*ones(NE,1); % weight of 1/3-v1
25    1/3*ones(NE,1); % weight of 1/3-v2
26    1/3*ones(NE,1); % weight of 2/3-v1
27    2/3*ones(NE,1); % weight of 2/3-v2
28    1/3*ones(3*NT,1) % center-v1,v2,v3
29    ];
30 Pro{J} = sparse(ii,jj,ss,N+2*NE+NT,N);
31 Res{J} = Pro{J}';

```

13.7.3 高次元 MG 使用线性元传递的原因

对高阶元, 嵌套关系如下

$$V_1 \subset V_2 \subset \cdots \subset V_L \subset V_h, \quad L = J,$$

这里 V_L 和 V_h 都位于最高层网格上, 而后者是高阶元空间.

- 如果去掉 V_1, \dots, V_L 对应的这些步骤, 那么 MG 实际上是对高阶元空间 V_h 的问题使用经典迭代法求解.
- 由于经典迭代法对残差的低频分量的抑制作用有限, 在经典迭代的基础上, MG 又添加了 V_1, \dots, V_L 对应的步骤, 用以快速消除低频分量.
- 正因为有 V_h 这一步的精度保障, V_1, \dots, V_L 的步骤使用线性元即可.

- 注意, 尽管 MG 是在经典迭代法的过程中添加了额外的步骤, 但计算量并没有增加.
原因是这些额外步骤会显著地减少迭代步数.

第十四章 自适应网格上的多重网格法

14.1 最新点二分加密的网格粗化算法

参考文献

- [1] K. Feng and Z.C. Shi. Mathematical Theory of Elastic Structures [M]. Springer-Verlag, 1996.
- [2] Brenner, Susanne C. Forty years of the Crouzeix-Raviart element. Numer. Methods Partial Differential Equations. 31(2), 367-396, 2015.
- [3] Bahriawati, C. and Carstensen, C. Three MATLAB implementations of the lowest-order Raviart-Thomas M-FEM with a posteriori error control. Comput. Methods Appl. Math. 5(4), 333-361, 2005.
- [4] Beirão da Veiga, L., Niiranen, J. and Stenberg, R. A posteriori error estimates for the Morley plate bending element. Numer. Math. 106(2), 165-179, 2007.
- [5] S. C. Brenner and L. Y. Sung. Linear finite element methods for planar linear elasticity. Math. Comp. 59(200), 321-338, 1992.
- [6] Falk, Richard S. Nonconforming finite element methods for the equations of linear elasticity. Math. Comp. 57(196), 529-550, 1991.
- [7] P. O. Persson and G. Strang. A simple mesh generator in Matlab. SIAM Rev. 46(2), 329–345, 2004.